

OSP-10546 us

日 本 国 特 許 庁  
PATENT OFFICE  
JAPANESE GOVERNMENT

US  
Jc997 U.S. PTO  
09/837731

別紙添付の書類に記載されている事項は下記の出願書類に記載されて  
いる事項と同一であることを証明する。

This is to certify that the annexed is a true copy of the following application as filed  
with this Office.

出 願 年 月 日

Date of Application:

2000年 4月20日

出 願 番 号

Application Number:

特願2000-119594

出 願 人

Applicant(s):

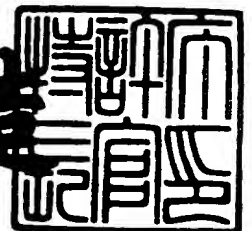
日本電気株式会社

CERTIFIED COPY OF  
PRIORITY DOCUMENT

2001年 2月16日

特許庁長官  
Commissioner,  
Patent Office

及川耕造



【書類名】 特許願

【整理番号】 37300360

【特記事項】 特許法第 3 6 条の 2 第 1 項の規定による特許出願

【提出日】 平成12年 4月20日

【あて先】 特許庁長官 殿

【国際特許分類】 G06F 9/44  
G06F 9/45

【発明者】

【住所又は居所】 東京都港区芝五丁目 7 番 1 号 日本電気株式会社内

【氏名】 マイルス ゴードン ベーダー

【特許出願人】

【識別番号】 000004237

【氏名又は名称】 日本電気株式会社

【代理人】

【識別番号】 100088328

【弁理士】

【氏名又は名称】 金田 暢之

【電話番号】 03-3585-1882

【選任した代理人】

【識別番号】 100106297

【弁理士】

【氏名又は名称】 伊藤 克博

【選任した代理人】

【識別番号】 100106138

【弁理士】

【氏名又は名称】 石橋 政幸

【手数料の表示】

【予納台帳番号】 089681

【納付金額】 35,000円

【提出物件の目録】

【物件名】 外国語明細書 1

【物件名】 外国語図面 1

【物件名】 外国語要約書 1

【包括委任状番号】 9710078

【プルーフの要否】 要

【書類名】 外国語明細書

1. Title of Invention

A method for avoiding excessive overhead while using a form of SSA (Static Single Assignment) extended to use storage locations other than local variables.

2. Claims

(1) A method for avoiding excessive overhead by a programmed computer while using a form of SSA (Static Single Assignment) extended to use storage locations other than local variables, comprising the step of:

allowing a program to use a compiler representation known as SSA form on any memory location addressable by the program; SSA form is normally only usable on function local variables.

(2) The method as claimed in claim 1, further comprising the steps of:

inserting phi functions at any place in the function where multiple definitions of a same non-SSA variable may be merged, the phi-functions producing a new definition of the variable at a point where they are inserted;

finding which operations may implicitly read or write complex variables that are in SSA form;

adding write-back copy operations at appropriate locations to write complex variables that are in SSA form, the write-back copy operations writing an SSA variable back to its real location;

adding read-back copy operations at appropriate locations to read possibly modified values back into new SSA definitions, the read-back copy operations defining a new SSA variable from a variable's real location; and

replacing every non-SSA variable definition by a definition of a unique SSA-variable, and replacing every non-SSA variable reference by a reference to an appropriate SSA-variable.

### 3. Detailed Description of Invention

#### Technical Field to which the Invention pertains

The present invention relates to Compiler Optimization.

#### Prior Art

This technique is related to an extension of the usual formulation of Static Single Assignment (SSA) form.

Briefly, 'SSA form' is an alternative representation for variables in a program, in which any given variable is only assigned at a single location in the program. A program is transformed into SSA form by a process called 'SSA conversion'. SSA conversion replaces every local variable in the source program with a set of new variables, called 'SSA variables', each of which is only assigned to at a single physical location in the program -- thus, every point at which a source variable V is assigned to in the source program, the corresponding SSA-converted program will instead assign a unique variable, V'1, V'2, etc. At any point in the program (always at the start of a basic-block) where the merging of control flow would cause two such derived variables to be live simultaneously, their values are merged together to yielding a single new SSA variable, e.g., V'3, that represents the value of the original source variable at that point. This merging is done using a 'phi-function'. A phi-function is an instruction which has as many inputs as there are basic-blocks that can transfer control to the basic-block it is in, and chooses whichever input corresponds to the basic-block that preceded the current one in the dynamic control flow of the program.

SSA form is convenient because it allows variables to be treated as values, independent of their location in the program, making many transformations more straight-forward, as they don't need to worry about the implicit constraints imposed by using single variable

names to represent multiple values, depending on the location in the program. These properties make it a very useful representation for an optimizing compiler -- many optimization techniques become significantly simpler if the program is in SSA form. For instance, in a traditional compiler, a simple common-sub-expression-elimination algorithm that operates on variables must carefully guard against the possibility of redefinition of variables, so that it generally is only practical to use within a single basic block. However, if the program is in SSA form, this simple optimization need not worry about redefinition at all -- variables can't be redefined -- and furthermore will work even across basic block boundaries.

SSA conversion, as described above, is a transformation that is traditionally applied only to a function's local variables; this makes the process much easier, as local variables are subject to various constraints. For instance one knows that local variables are not aliased to other local variables, and unless its address has been taken, that a local variable will not be modified by a function other than the one it is declared in.

However, there are many cases where 'active values', which one would like to receive the benefits of optimizations made possible by using SSA-form, exist in storage locations other than local variables. In this case, one would like to have the object's fields receive the same treatment as if they were a local variable, which could yield optimizations.

Information about SSA form can be found in the paper:

[SSAFORM] 'Efficiently Computing Static Single Assignment Form and the Control Dependence Graph', by Ron Cytron et al., ACM TOPLAS, Vol. 13, No. 4, October 1991, Pages 451-490.

The SSA-conversion process in [SSAFORM] is performed in two steps [figure 12]:

- (a) [201] Phi functions are inserted at any place in the function where multiple

definitions of the same non-SSA variable may be merged. The phi-functions produce a new definition of the variable at the point where they are inserted.

Because of this step, there is only one extant definition of a source variable at any point in the program.

(b) [202] Every non-SSA variable definition is replaced by a definition of a unique SSA-variable, and every non-SSA variable reference replaced by a reference to an appropriate SSA-variable -- because of the insertion of phi-functions, there will always be a single extant SSA-variable corresponding to a given non-SSA variable.

An extension of SSA form to non-local locations is described in:

[SSAMEM] 'Effective Representation of Aliases and Indirect Memory Operations in SSA Form', by Fred Chow et al., Lecture Notes in Computer Science, Vol. 1060, April 1996, Pages 253-267.

The concept of basic-block 'dominance' is well known, and can be described as follows:

A basic block A 'dominates' a basic block B, if the flow of control can reach B only after A (although perhaps not immediately; other basic blocks may be executed between them).

If A dominates B and no other block dominates B that doesn't also dominate A, then A is said to be B's 'immediate dominator'.

#### Problem to be solved by the Invention

It is desirable to extend the use of SSA form to handle non-local memory locations. However, a straight-forward implementation given the prior art, which synchronizes SSA

representations at every point of unknown behavior, can be very inefficient, because there are many operations that may read or write almost *\*any\** memory location (for instance, in the case of library function calls, where the compiler often has no information about their behavior). Using such a simple technique also causes many extra phi-functions to be introduced, which can dramatically increase the cost of using SSA form.

This invention attempts to use SSA form on non-local memory locations, without excessive overhead for common program structures, by consolidating memory synchronization operations where possible.

#### Means for solving Problem

In this invention, we modify the procedure of [SSAFORM][figure 12] as follows:

Method for representing pointer variables in SSA form[452]:

+ References or definitions of memory locations resulting from pointer-dereferences are also treated as 'variables', here called 'complex variables' [452], in addition to simple variables [451], such as those used in the source program. Complex variables consist of a pointer variable and an offset from the pointer. An example of a complex variable is the C source expression (lvalue) '\*p', as used in [810] and [820].

Method for adding appropriate copy operations to synchronize complex variables [452] with the memory location they represent [figure 1]:

+ These 'complex variables' [452] are treated as non-SSA variables during SSA-conversion [figure 1] (any variable reference within a complex variable is treated as a reference in the instruction [440] that contains the complex variable [452]).



+ A new step [120] is inserted in the SSA-conversion process [figure 1] between steps (a) [110] and (b) [130], to take care of any necessary synchronization of SSA-converted complex variables [452] with any instructions [440] that have unknown side-effects:

(a') [121] To any instruction [440] that may have unknown side-effects on an 'active' complex variable [452] -- one that is defined by some dominator of the instruction -- add a list of the variable, and the possible side effects (may\_read, may\_write).

[122, 123] Next, insert special copy operations, called write-backs [521] (which write an SSA variable back to its real location) and read-backs (which define a new SSA variable from a variable's real location), to make sure the SSA-converted versions of affected variables [450] correctly synchronized with respect to such side-effects. This step may also insert new phi-functions, in the case where copying back a complex variable [452] from it's synchronization location may define a new SSA version of that variable.

For an example of adding write-backs [521] and read-backs [522], see [figure 4].

#### Mode for carrying out the Invention

This invention is an addition to a compiler for a computer programming language, whose basic control flow is illustrated in [figure 2]:

A source program [301] is converted into an internal representation by a parser [310], and if optimization is enabled, the internal representation is optimized by the optimizer [320]. Finally, the internal form is converted into the final object code [302] by the backend [330]. In a compiler that uses SSA form, the optimizer usually contains at least three steps: conversion of the program from the 'pre-SSA' internal representation into an internal representation that uses SSA form [figure 1], optimization of the program in SSA form [322], and conversion of the program from SSA form to an internal representation without SSA form [323]. Usually SSA form differs from the non-SSA internal

representation only in the presence of additional operations, and certain constraints on the representation; see [SSAFORM] for details.

The preferred internal representation of a program used is as follows [figure 3]:

A program [410] is a set of functions.

A function [420] is a set of 'blocks' [430], roughly corresponding to the common compiler concept of a 'basic block'. A flow graph is a graph where the vertices are blocks [430], and the edges are possible transfers of control-flow between blocks [430]. A single block [430] is distinguished as the 'entry block' [421], which is the block in the function executed first when the function is called.

Within a block [430] is a sequence of 'instructions' [440], each of which describes a simple operation. Within a block [430], control flow moves between instructions [440] in the same order as their sequence in the block; conditional changes in control flow may only happen by choosing which edge to follow when choosing the successor block [432] to a block, so if the first instruction [440] in a block is executed, the others are as well, in the same sequence that they occur in the block [430].

An instruction [440] may be a function call, in which case it can have arbitrary side-effects, but control-flow must eventually return to the instruction [440] following the function call.

An instruction [440] may explicitly read or write 'variables' [450], each of which is either a 'simple variable' [451], such as a local or global variable in the source program (or a temporary variable created by the compiler), or a 'complex variable' [452], which represents a memory location that is indirectly referenced through another variable.

Each variable has a type, which defines what values may be stored in the variable.

Complex variables [452] are of the form  $\text{*(BASE + OFFSET)}$ , where BASE [453] is a variable [450], and OFFSET [454] is a constant offset; this notation represents the value stored at memory location (BASE + OFFSET).

Because of the use of complex variables [452], there are typically no instructions [440] that serve to store or retrieve values from a computed memory location. Instead, a simple copy where either the source or destination, or both, is a complex variable [452] is used. Similarly, any other instruction [440] may store or retrieve its results and operands from memory using complex variables.

To assist in program optimization, each function is converted to SSA-form, which is described in (Prior Art) section, as modified for this invention, described in (Means for solving Problem). section. This conversion is called SSA-conversion, and takes place in 3 steps [figure 1], (a), (a'), and (b):

(a) [110] Phi functions are inserted at any place in the function where multiple definitions of the same variable may be merged, as described in [SSAFORM]. The phi-functions produce a new definition of the variable at the point where they are inserted. For example, the phi function [910] is inserted to merge the different values written to the complex variable  $\text{*p}$  at [911] ([820] in the input program) and [912] ([830] in the input program), and also at [1010], merging the values defined at [1011] ([820] in the input program) and [830] in the input program.

Because of this step, there is only one extant definition of a source variable at any point in the program.

(a') I. [121] For each operation, determine which 'active' complex variables [452] it may

have unknown side-effects on, and list attach a note to the operation with this information. These notes are referred to below as 'variable syncs'. In the example program, instructions [1020], [1021], [1022], and [1023] may possibly read or modify '\*p' (as we don't have any information about them).

II. [122] At the same time, add any necessary write-back copy operations [521] write back any complex variables [452] to their 'synchronization location'-- which is the original non-SSA variable (which, for complex variables [452], is a memory location), and mark the destination of the copy operation as such (this prevents step (b) of SSA conversion from treating the destination of the copy as a new SSA definition). Any such 'write-back' [521] makes the associated variable inactive, and so prevents any further write-backs [521] unless the variable is once again defined.

III. [123] Add necessary read-backs, to supply new SSA definitions of complex variables [452] that have been invalidated (after having been written back to their synchronization location).

This is done by essentially solving a data-flow problem, where the values are 'active read-backs', which are:

- + Defined by operations that may modify a complex variable [452], as located in step I above, or by the merging of multiple active read-backs of the same variable [450], at control-flow merge points. In the example, all the function calls may possibly modify '\*p', so they must be represented by read-backs at [1020], [1021], [1022], and [1024].

- + Referenced by operations that use the value of a complex variable with an active read-back, or reaching a control-flow merge point at which no other read-backs of that variable are active (because such escaped definitions must then be merged with any other values of the complex variable using a phi-function).

Only read-backs that are referenced must actually be instantiated. In the example program, the only instantiated read-back is at [1030]. The reference that causes instantiation is the assignment of `*p` to the variable `x`, at [840] in the source program; in the SSA-converted program, this assignment is split between the read-back at [1030] and the phi function at [1031].

+ Killed by definitions of the associated complex variable [452], or by a new read-back of the variable. In the example, the read-back defined at [1021] is killed because the following function call defines a new read-back of the same variable at [1022].

+ Merged, at control-flow merge points, with other active read-backs of the same variable [450], resulting in a new active read-back of the same variable. In the example, a 'merge read-back' is defined at [1030], merging the read-backs of `*p` at [1022] and [1023].

After a fixed-point of read-back definitions is reached, those that are referenced are instantiated by inserting the appropriate copy operation at the place where they are defined, to copy the value from the read-back variable [450]'s synchronization location into a new SSA variable; if necessary new phi-functions may be inserted to reflect this new definition point. As

mentioned above, in the example this only happens at [1030].

Steps (a'.I) [121] and (a'.II) [122] take place as follows:

Call the procedure `'add_syncs_and_write_backs'` [figure 5] on the function's entry block [430], initializing the `ACTIVE_VARIABLES` and `ALL_ACTIVE_VARIABLES` parameters to empty lists.

The procedure ``add_syncs_and_write_backs'`, with arguments `BLOCK`, `ACTIVE_VARIABLES`, and `ALL_ACTIVE_VARIABLES` is defined as follows [figure 5]:

[610] For every instruction [440] in the `BLOCK`, do:

[620] For each `VARIABLE` in `ALL_ACTIVE_VARIABLES`, do:

[621] If `INSTRUCTION` may possibly read or write `VARIABLE`, then [622] add a ``variable sync'` describing the possible reference or modification to `INSTRUCTION`.

[625] If `INSTRUCTION` may possibly read or write `VARIABLE`, and is also in `ACTIVE_VARIABLES`, then [626] add a ``write-back'` copy operation just before `INSTRUCTION` to write `VARIABLE` back to its synchronization location, and [627] remove `VARIABLE` from `ACTIVE_VARIABLES`. Because at this stage of SSA conversion, only source variables are present (not SSA variables), then this write-back copy operation is represented by a copy from `VARIABLE` to itself (``VARIABLE := VARIABLE'`) with a special flag set to indicate that the destination should not be SSA-converted.

[630] For each `VARIABLE` which is defined in `INSTRUCTION`, do:

Add `VARIABLE` to `ACTIVE_VARIABLES` and `ALL_ACTIVE_VARIABLES` (modifications to these variables are local to this function).

[650] For each block [430] immediately dominated by `BLOCK`, `DOM`, do:

[651] Recursively use `add_syncs_and_write_backs` on the dominated block `DOM`, with the local values of `ACTIVE_VARIABLES` and `ALL_ACTIVE_VARIABLES` passed as

the respectively named parameters.

Step (a'.III) take place as follows [figure 6]:

[701] Initialize the mappings BLOCK\_BEGIN\_READ\_BACKS and BLOCK\_END\_READ\_BACKS to be empty. These mappings associate each block in the flow graph with a sets of read-backs.

[702] Initialize the queue PENDING\_BLOCKS to the function's entry block.

[710] While PENDING\_BLOCKS is not empty, [711] remove the first block [430] from it, and invoke the function 'propagate\_block\_read\_backs' [800] on that block.

[720] For each read-back RB in any block [430] that has been marked as 'used', and [721] isn't a 'merge read-back' who's sources (the read-backs that it merges) are all also marked 'used', instantiate that read-back as follows:

[730] If RB is a 'merge read-back', then the point of read-back is [741] the beginning of the block [430] where the merge occurs, otherwise it is [742] immediately after the instruction [440] that created the read-back.

[731] Add a copy operation at the point of read-back that copies RB's variable from its synchronization location to an SSA variable (as noted above for adding write-back copy operations, because at this stage no SSA variable have actually been introduced, this copy operation simply copies from the variable to itself, but marks the source of the copy with a flag saying not to do SSA conversion).

[732] If necessary, introduce phi functions to merge the newly defined SSA variable with other definitions of the variable.

The function 'propagate\_block\_read\_backs', with the parameter BLOCK, is defined as follows [figure 7]:

[801] Look up BLOCK in BLOCK\_BEGIN\_READ\_BACKS and BLOCK\_END\_READ\_BACKS, assigning the associated read-back set with the local variables OLD\_BEGIN\_READ\_BACKS and OLD\_END\_READ\_BACKS respectively. If there is no entry for block in either case, add an appropriate empty entry for block.

[810] Calculate the intersection of the end read-back sets for each predecessor block [431] of BLOCK in the flow-graph, calling the result NEW\_BEGIN\_READ\_BACKS. The intersection is calculated as follows:

Any predecessor read-back for which a read-back of the same variable doesn't exist in one of the other predecessor blocks is discarded from the result; it is also marked as 'referenced'.

If the read-back for a given variable is the same read-back in all predecessor blocks [431], that read-back is added to the result.

If a given variable is represented by different read-backs in at least two predecessor blocks [431], a 'merge read-back' is created that references all the corresponding predecessor read-backs, and this merge read-back is added to the result.

[820] If NEW\_BEGIN\_READ\_BACKS is different from OLD\_BEGIN\_READ\_BACKS, or this is the first time this block has been processed, then:

[821] Add NEW\_BEGIN\_READ\_BACKS as the entry for BLOCK in



BLOCK\_BEGIN\_READ\_BACKS, replacing OLD\_BEGIN\_READ\_BACKS.

[822] Initialize NEW\_END\_READ\_BACKS from NEW\_BEGIN\_READ\_BACKS.

[830] For each operation INSTRUCTION in BLOCK, do:

[840] For each variable reference VREF in INSTRUCTION, do:

[845] If VREF has an entry RB in NEW\_END\_READ\_BACKS, then [846] Mark RB as used, and [847] remove it from NEW\_END\_READ\_BACKS.

[850] For each variable definition VDEF in INSTRUCTION, do:

[855] If VDEF has an entry RB in NEW\_END\_READ\_BACKS, then [856] remove RB from NEW\_END\_READ\_BACKS.

[860] For each variable sync in INSTRUCTION that notes a variable VARIABLE as possibly written, do:

[865] Add a new read-back entry for VARIABLE to NEW\_END\_READ\_BACKS, replacing any existing read-back of VARIABLE.

[870] If NEW\_END\_READ\_BACKS is different from OLD\_END\_READ\_BACKS, then:

[871] Add NEW\_END\_READ\_BACKS as the entry for BLOCK in BLOCK\_END\_READ\_BACKS, replacing OLD\_END\_READ\_BACKS.

[880] Add each BLOCK's successors [432] to PENDING\_BLOCKS.

(b) [130] Every non-SSA variable definition is replaced by a definition of a unique SSA-variable, and every non-SSA variable reference replaced by a reference to an appropriate SSA-variable, as described in [SSAFORM].

The exception to this rule is complex variables [452] that have been marked as special 'synchronization' locations, in the copy instruction [440] inserted in step (a'); they are left as-is, referring to the original complex variable [452].

An example of a program being transformed into SSA form, with and without the use of this invention, can be found in figures 8 - 11.

#### Effect of the Invention

This invention adds synchronization operations that allow the efficient use of SSA-form for non-local memory locations in the presence of possible aliasing.

#### 4. Brief Description of Drawings

Figure 1 shows general form of the SSA-conversion process used by this invention.

Figure 2 shows overall compiler control flow.

Figure 3 shows basic data structures used in describing this invention.

Figure 4 shows placement of variable read- and write-backs.

Figure 5 shows the control flow of the procedure for steps (a'.I) and (a'.II) of the modified SSA conversion process, adding variable synchronization information to instructions and adding variable write-backs to a function, 'add\_syncs\_and\_write\_backs'.

Figure 6 shows the control flow of the procedure for step (a'.III) of the modified SA conversion process, adding variable read-backs to a function.

Figure 7 shows the control flow for a subroutine used by step (a'.III) of the modified SSA conversion process, 'add\_merged\_read\_backs'.

Figure 8 shows example source program.

Figure 9 shows SSA converted program, with simple implementation of read-backs.

Figure 10 shows SSA converted program, with the implementation of read-backs described in this patent.

Figure 11 shows register-allocated and SSA-unconverted program.

Figure 12 shows general form of the traditional SSA-conversion process.

Description of the Reference Numerals

- 301 Input source file
- 320 Optimizer
- 302 Output file
- 410 Program
- 420 Function
- 421 Entry block
- 430 Block
- 440 Instruction
- 450 Variable

【書類名】 外国語図面

Figure 1. Modified SSA-conversion process

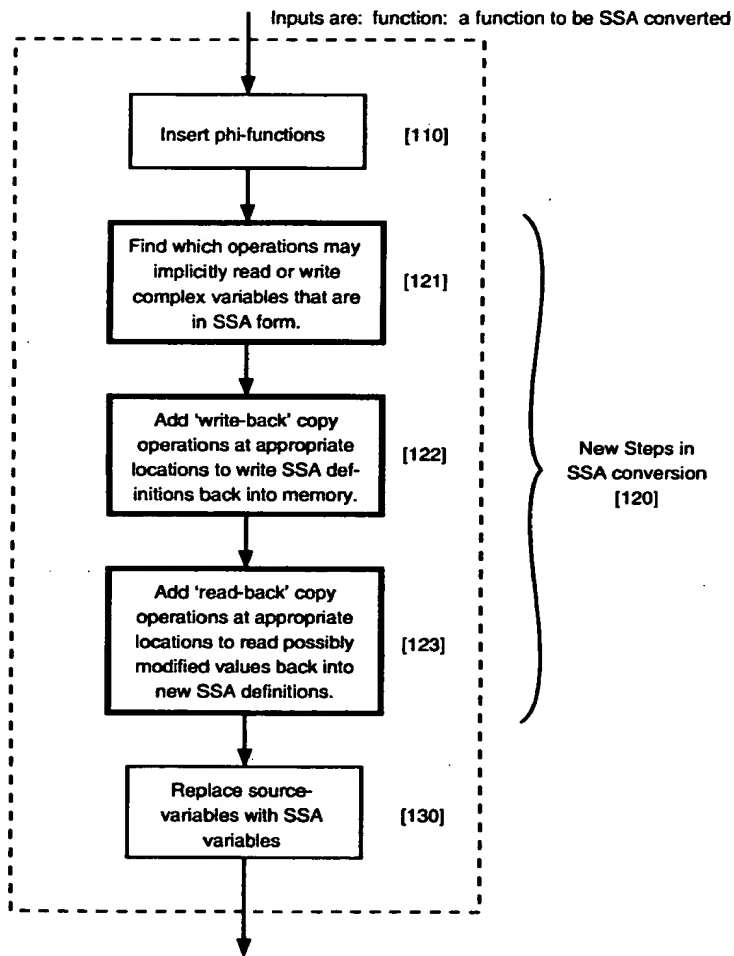


Figure 2. Overall compiler control flow

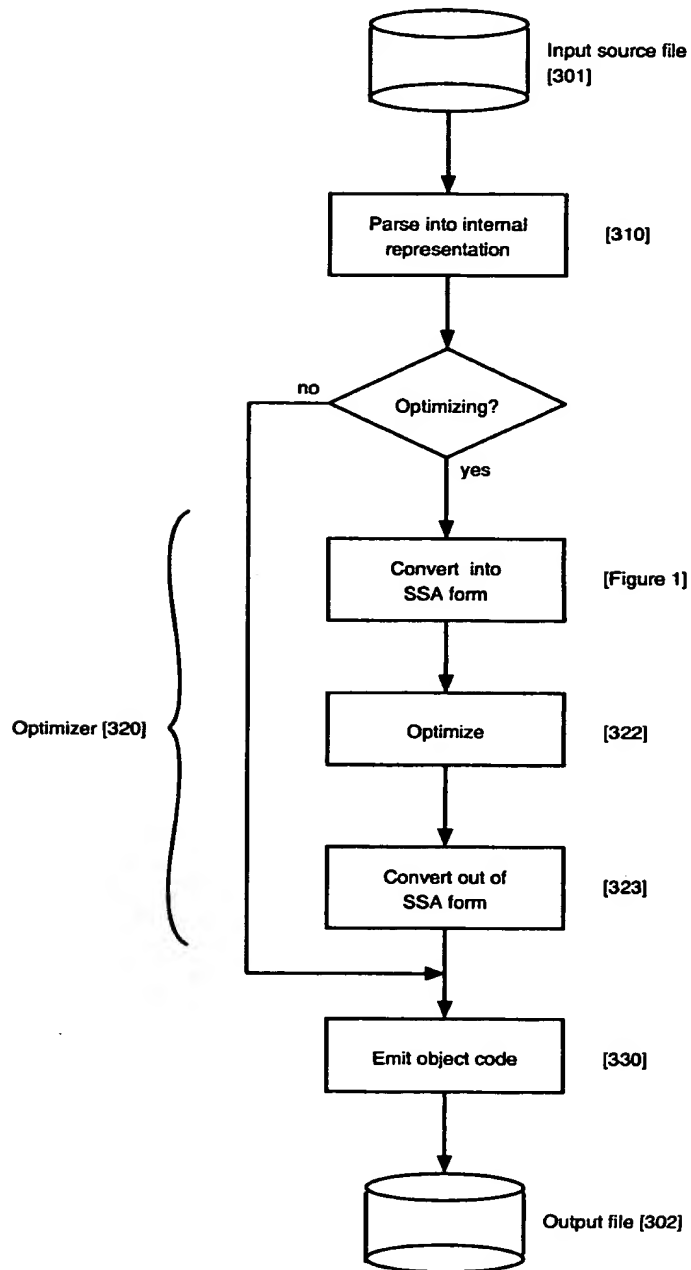


Figure 3. Program representation

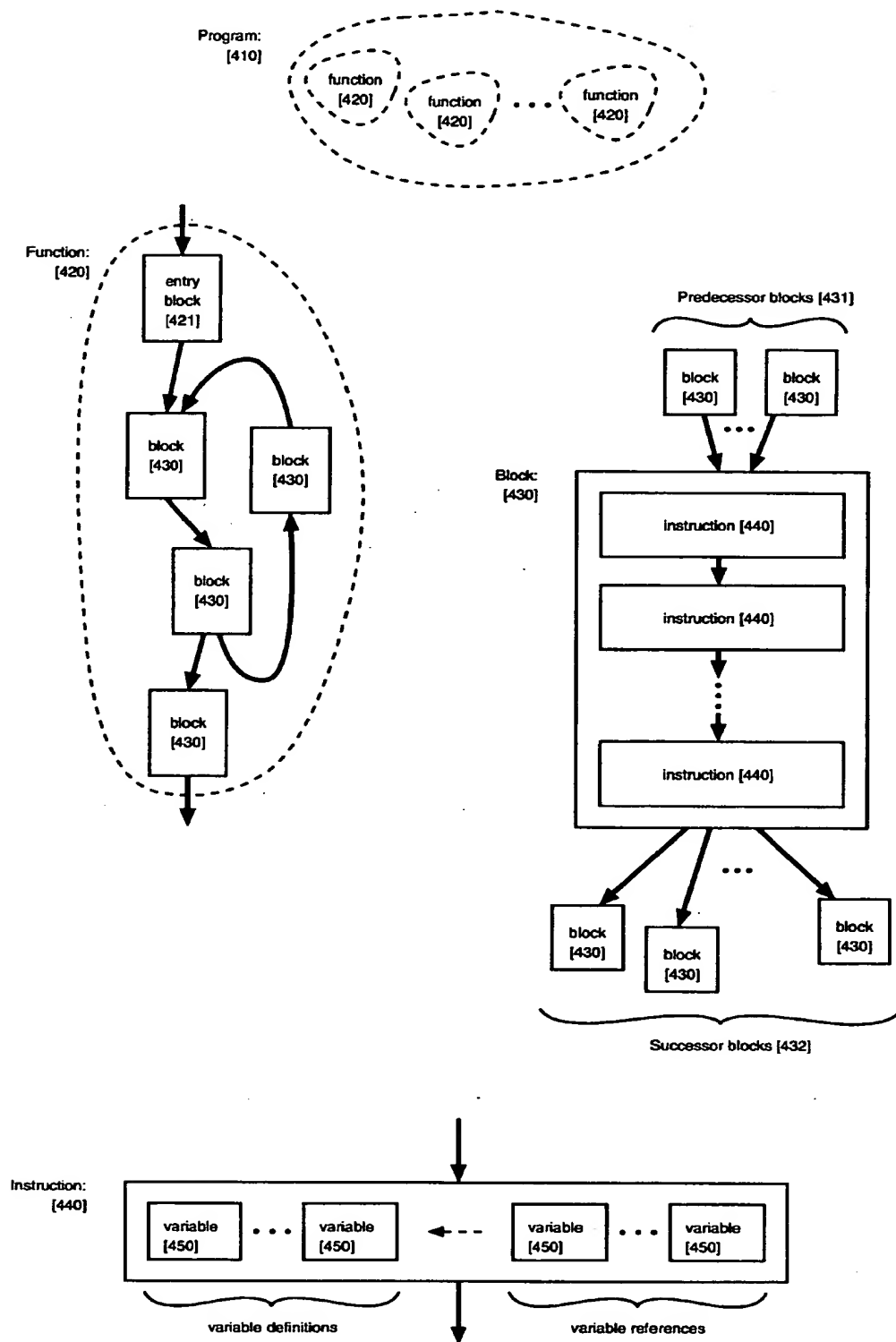


Figure 4. Placement of read/write-backs for the SSA form of  $*x$ ,  $(*x)'1$

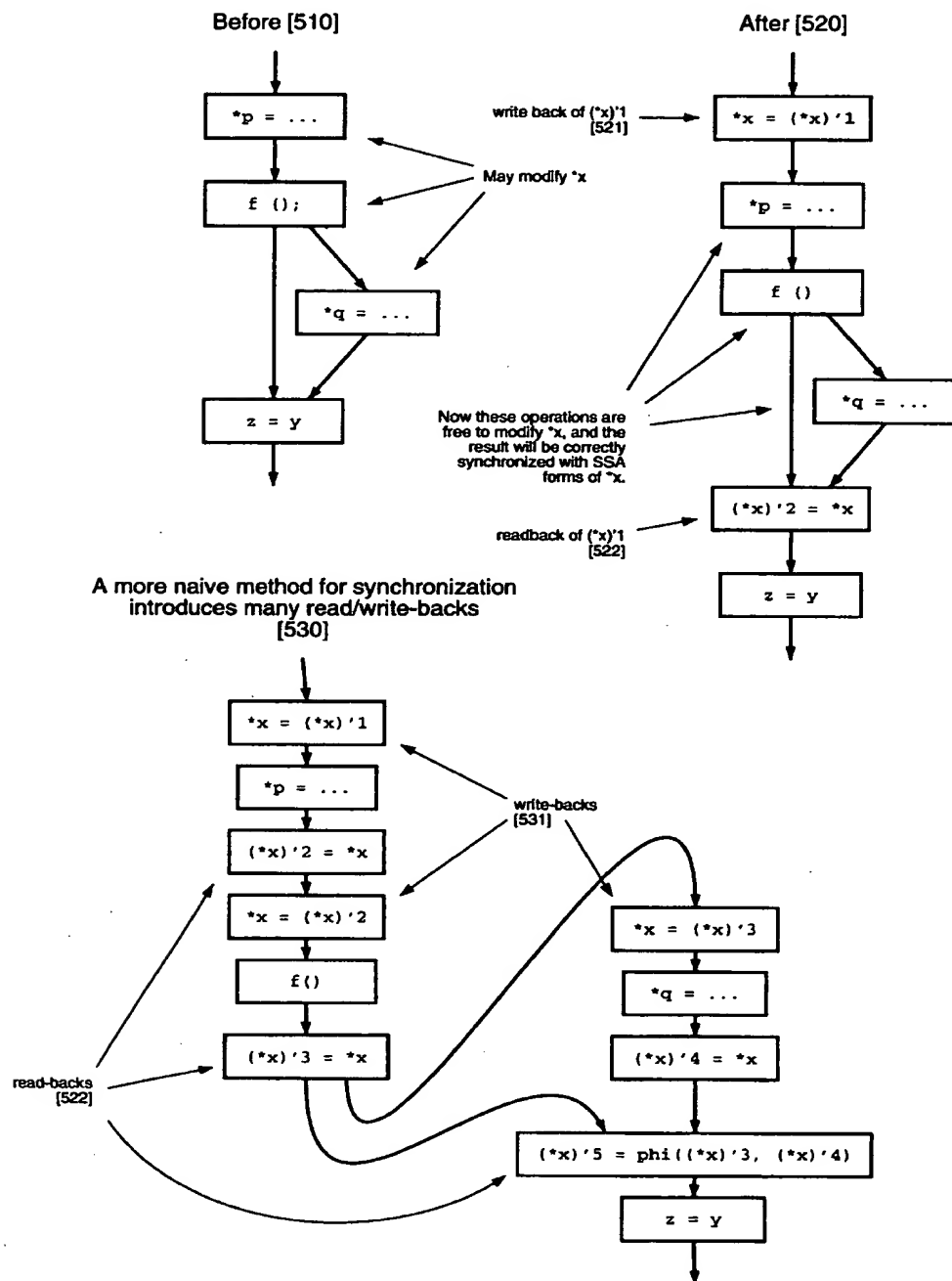


Figure 5. The procedure 'add\_syncs\_and\_write\_backs'

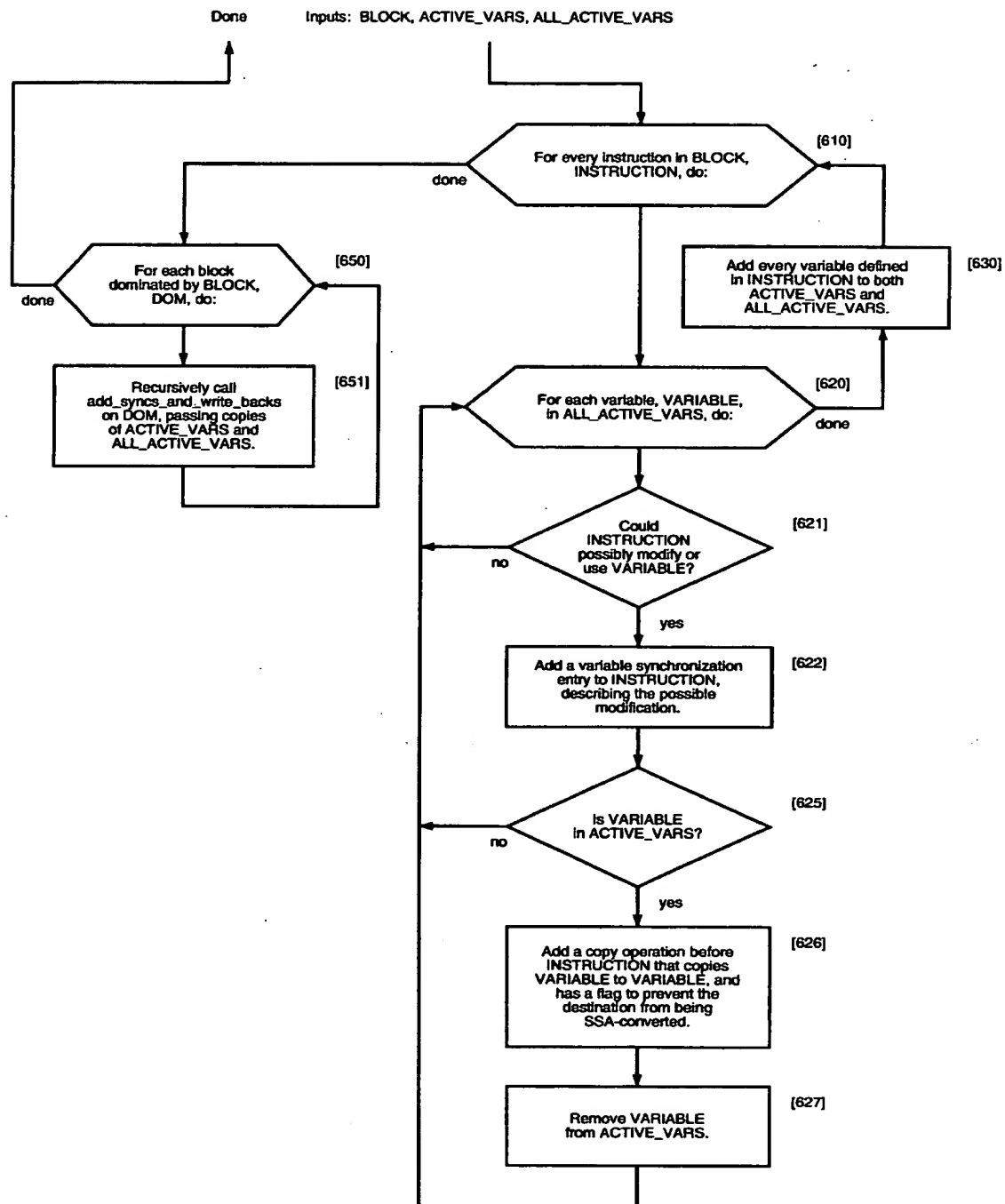




Figure 6. Conversion step (a'.III), insertion of read-backs

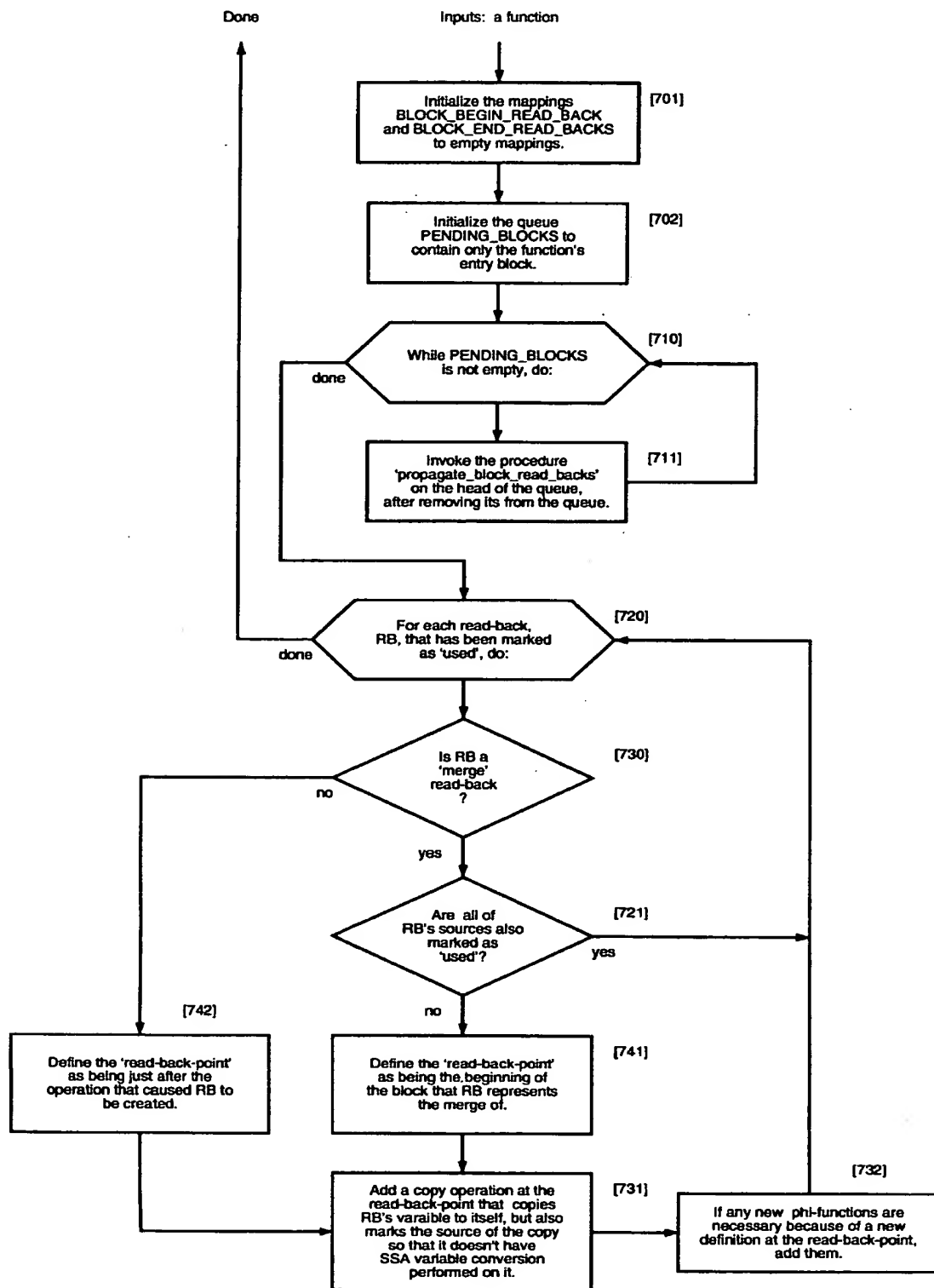


Figure 7. The procedure 'propagate\_read\_backs'

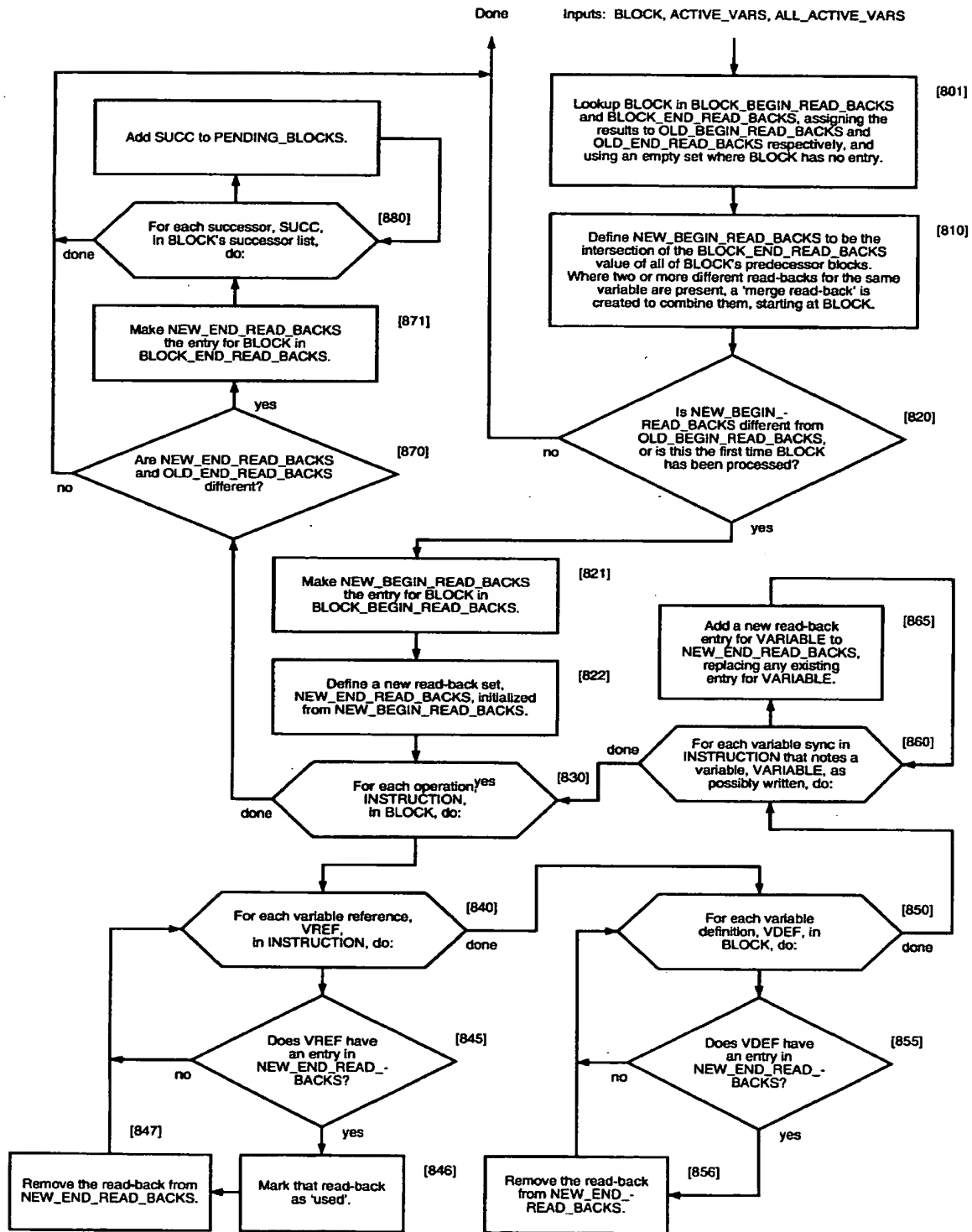


Figure 8: Example source program

This short C program is used to illustrate the invention:

```

extern int g (), h (), i (), x;
int foo (int *p)
{
    (*p)++;
    if (*p > 10)
    {
        g ();
        h ();
        if (x > 5)
            g ();
        if (x > 3)
            i ();
        else
            x = *p;
        *p = 5;
    }
    return *p;
}

```

[810]

Here's the same program converted to a slightly more primitive form:

```

int foo (int *p)
{
    block1:
        *p := *p + 1;
        if (*p <= 10)
            goto block8;

    block2:
        g ();
        h ();
        if (x <= 5)
            goto block4;

    block3:
        g ();

    block4:
        if (x > 3)
            goto block6;

    block5:
        x := *p;
        goto block7;

    block6:
        i ();

    block7:
        *p := 5;

    block8:
        return *p;
}

```

[820]

[840]

[830]

Figure 9: SSA converted program, with simple implementation of read-backs:

The following is psuedo-C, augmented with the 'phi' operation, where

```
RESULT = phi (block1: VAL1, ..., blockN: VALN)
```

means 'assign VAL1 to RESULT if control-flow comes from block1', and similarly so on for each value of N.

The extra variables "pvN", where N is an integer, are SSA versions of \*p, and are in fact local variables, not dereferences of p.

```
int foo (int *p)
{
    int pv1, pv2, pv3, pv4, pv5, pv6;

    block1:
        pv1 = *p + 1;
        if (pv1 <= 10)
            goto block8;

    block2:
        *p = pv1;          /* This writes-back PV1 to *P. */
        g ();
        pv2 = *p;          /* This reads-back *P into PV2. */
        *p = pv2;          /* This writes-back PV2 to *P. */
        h ();
        pv3 = *p;          /* This reads-back *P into PV3. */      [912]
        if (x <= 5)
            goto block4;

    block3:
        *p = pv3;          /* This writes-back PV4 to *P. */
        g ();
        pv4 = *p;          /* This reads-back *P into PV4. */      [911]

    block4:
        pv5 = phi (block3: pv4, block2: pv3)      [910]
        if (x > 3)
            goto block6;

    block5:
        goto block7;

    block6:
        i ();

    block7:
        x = phi (block6: x, block5: pv5);

    block8:
        pv6 = phi (block1: pv1, block7: 5);
        *p = pv6;          /* This writes-back PV6 to *P. */

    return pv6;
}
```

Figure 10: SSA converted program, with the implementation of read-backs described in this patent

```

int foo (int *p)
{
    int pv1, pv2, pv3;

    block1:
        pv1 = *p + 1;                                [1011]
        if (pv1 <= 10)
            goto block8;

    block2:
        *p = pv1;          /* This writes-back PV1 to *P.  */
        g ();                                [1021]
        h ();                                [1022]
        if (x <= 5)
            goto block4;

    block3:
        g ();                                [1023]

    block4:
        pv2 = *p;          /* This reads-back *P into PV2.  */
        if (x > 3)
            goto block6;                                [1030]

    block5:
        goto block7;

    block6:
        i ();                                [1024]

    block7:
        x = phi (block6: x, block5: pv2);            [1031]

    block8:
        pv3 = phi (block1: pv1, block7: 5);
        *p = pv3;          /* This writes-back PV3 to *P.  */
        return pv3;
}

```

Figure 11: Register-allocated and SSA-unconverted program

Using SSA-form requires having a good register allocator that will merge variables where possible, as it tends to generate a lot of variables with short lifetimes. We assume that here.

```

int foo (int *p)
{
    int pv;

block1:
    pv = *p + 1;
    if (pv <= 10)
        goto block8;

block2:
    *p = pv;          /* This writes-back PV to *P. */
    g ();
    h ();
    if (x <= 5)
        goto block4;

block3:
    g ();

block4:
    if (x > 3)
        goto block6;

block5:
    x = *p;
    goto block7;

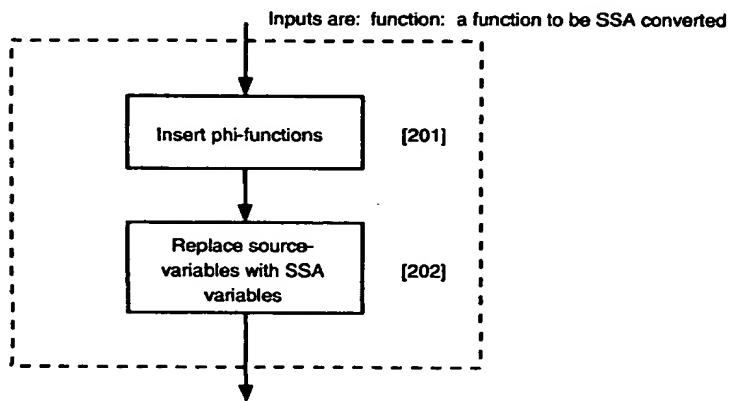
block6:
    i ();

block7:
    pv = 5;

block8:
    *p = pv;          /* This writes-back PV to *P. */
    return pv;
}

```

Figure 12. Original SSA-conversion process



【書類名】 外国語要約書

1. Abstract

subjective

The usual formulation of the compiler representation known as 'SSA-form' can only handle local variables. It is desirable to extend this to allow other locations to be represented.

means for solution

This invention adds synchronization operations that allow the efficient use of SSA-form for non-local memory locations in the presence of possible aliasing.

2. Representative Drawing

Figure 1



【書類名】 翻訳文提出書

【整理番号】 37300360

【提出日】 平成12年 6月 9日

【あて先】 特許庁長官 殿

【出願の表示】

    【出願番号】 特願2000-119594

【特許出願人】

    【識別番号】 000004237

    【氏名又は名称】 日本電気株式会社

【代理人】

    【識別番号】 100088328

    【弁理士】

    【氏名又は名称】 金田 暢之

    【電話番号】 03-3585-1882

【確認事項】 本書に添付した翻訳文は、外国語書面出願の願書に添付して提出した外国語明細書、外国語図面及び外国語要約書に記載した事項を過不足なく適正な日本語に翻訳したものである。

【提出物件の目録】

    【物件名】 外国語明細書の翻訳文 1

    【物件名】 外国語図面の翻訳文 1

    【物件名】 外国語要約書の翻訳文 1

【プルーフの要否】 要

【書類名】 明細書

【発明の名称】 局所変数以外の記憶位置を用いるように拡張された S S A 形式を使用して、過度のオーバーヘッドを避ける方法

【特許請求の範囲】

【請求項 1】 S S A（静的単一割り当て）形式は関数の局所変数のみに通常は使用可能であるのに対し、プログラムが S S A 形式として知られるコンパイラ表現を、前記プログラムによりアドレス可能な任意の記憶場所に使用することを可能とする段階を有する、局所変数以外の記憶位置を用いるように拡張された S S A の形式を使用して、プログラムされたコンピュータにより過度のオーバーヘッドを避ける方法。

【請求項 2】 ファイ関数は前記ファイ関数が挿入された点に同一の非 S S A 変数の新しい定義を生成する関数であり、前記同一の非 S S A 変数の複数の定義をマージすることができる前記ファイ関数の任意の位置に前記ファイ関数を挿入する段階と、

どの操作が、S S A 形式であるコンプレックス変数を、暗黙的に読み取り又は書き込みすることができるかを検出する段階と、

ライトバック・コピー操作は S S A 変数をその実際の位置に書き戻す操作であり、S S A 形式であるコンプレックス変数を書き込むために適切な位置に前記ライトバック・コピー操作を追加する段階と、

リードバック・コピー操作は変数の実際の位置から新しい S S A 変数を定義する操作であり、新しい S S A 定義の中に修正された可能性のある値をリードバックするために適切な位置に前記リードバック・コピー操作を追加する段階と、

すべての非 S S A 変数定義をユニークな S S A 変数の定義により置換し、すべての非 S S A 変数レファレンスを適切な S S A 変数へのレファレンスにより置換する段階とをさらに有する請求項 1 記載の方法。

【発明の詳細な説明】

【0 0 0 1】

【発明の属する技術分野】

本発明はコンパイラの最適化に関する。

## 【 0 0 0 2 】

## 【従来の技術】

本技術は静的単一割り当て（SSA）形式の通常の公式化の拡張に関する。

## 【 0 0 0 3 】

簡潔に言えば、' SSA形式' はプログラムにおける変数に対する別の表現である。' SSA形式' において、変数はそのプログラム内で単一の位置にのみ割り当てられる。プログラムは' SSA変換' と呼ばれるプロセスにより SSA形式に変換される。SSA変換はソース・プログラム内のすべての局所変数を' SSA変数' と呼ばれる一組の新しい変数に置換する。' SSA変数' のそれぞれは、そのプログラム内で単一の物理的な位置、したがって、原変数Vがソース・プログラム内で割り当てられるすべての点にのみ割り当てられる。対応する SSA変換されたプログラムは、ユニークな変数V' 1、V' 2などを代わりに割り当てるであろう。制御流れのマーキングがこのような2つの導出された変数を同時に生変数とさせるプログラム内の任意の点（常に基本ブロックの開始点）において、2つの導出された変数の値は1つにマージされ、その時点で元の原変数の値を表す単一の新しい SSA変数、たとえば、V' 3を生ずる。このマーキングは' ファイ関数' を使用して行われる。ファイ関数は、命令がある基本ブロックに制御を転送することができる基本ブロックと同数の入力を有し、プログラムの動的な制御流れの中で現在の入力に先行する基本ブロックにどの入力に対応するかを選択する命令である。

## 【 0 0 0 4 】

SSA形式は、プログラム内の位置によって複数の値を表すために単一の変数名を使用することにより課される暗黙の制約条件を心配する必要がないので、プログラム内の変数の位置から独立して変数が値として処理されることを可能にし、多くの変換をより直接的にするので、SSA形式は都合が良い。これらの特性は最適化コンパイラにとって SSA形式を非常に有用な表現にする。プログラムが SSA形式ならば、多くの最適化技術は大幅に簡単になる。たとえば、従来のコンパイラでは、変数の上で操作する単純な共通式の削除アルゴリズムは、変数の再定義の可能性を注意深く保護しなければならない。したがって、従来のコン

パイラは単一の基本ブロックの中でのみ使用することが通常实际的である。しかし、プログラムがSSA形式であれば、この単純な最適化は再定義について心配する必要はまったくない。変数は再定義できず、さらに基本ブロック境界を越えて動作する。

## 【0005】

上述のように、SSA変換は関数の局所変数にのみ伝統的に応用されていた変換である。局所変数は様々な制約条件に支配されるから、SSA変換は処理を非常に容易にする。たとえば、局所変数は他の局所変数に別名を付けないことは知られており、局所変数のアドレスがとられないかぎり、局所変数はそれが宣言された関数以外の関数により修正されないことは知られている。

## 【0006】

しかし、SSA形式を使用することにより可能となる最適化の利益を受けるために望ましい、'アクティブな値'が局所変数以外の記憶位置に存在する多くの場合がある。この場合に、オブジェクトのフィールドが局所変数であるかのよう

## 【0007】

SSA形式についての情報は下記の論文に見いだすことができる。

## 【0008】

[SSAFORM]、'静的単一割り当て形式及び制御依存性グラフの効率的な計算'、ロン・サイトロン他、ACM TOPLAS、Vol. 13、No. 4、1991年10月、451-490ページ。

## 【0009】

[SSAFORM]内のSSA変換処理は2つの段階で行われる[図12]。

## 【0010】

(a) [201]、同一の非SSA変数の複数の定義をマージすることができる関数内の任意の位置にファイ関数が挿入される。ファイ関数は、ファイ関数が挿入される点に変数の新しい定義を作る。

## 【0011】

この段階のために、プログラム内の任意の点において、原変数の実在の定義は

1つのみ存在する。

【0012】

(b) [202]、ファイ関数の挿入のために、すべての非SSA変数の定義は固有のSSA変数の定義により置換され、すべての非SSA変数レファレンスは適切なSSA変数へのレファレンスにより置換される。一定の非SSA変数に対応して単一の実在のSSA変数が常に存在する。

【0013】

SSA形式の非局所的な位置への拡張は下記の文献に説明されている。

【0014】

[SSAMEM]、'SSA形式における別名と間接的なメモリ操作の有効な表現'、フレッド・チョー他、コンピュータ・サイエンス講義ノート、Vol. 1060、1996年4月、253-267ページ。

【0015】

基本ブロック'優位'の概念は公知であり、次のように説明できる。

【0016】

制御の流れがAの後にBに到達することができる場合にのみ、基本ブロックAは基本ブロックBの'優位に立つ'、(恐らく直ちにではないが、他の基本ブロックはそれらの間に実行されることができる)。

【0017】

AがBの優位に立ち、さらにAの優位に立たない他のどのブロックもBの優位に立たなければ、AはBの'イミディエイト・ドミネータ'である。

【0018】

【発明が解決しようとする課題】

非局所的記憶場所を処理するために、SSA形式の使用を拡張することが望ましい。しかし、未知の操作のすべての点においてSSA表現を同期させる従来技術による直接的な具体化例は、ほとんどすべての記憶場所を読み取りあるいは書き込みする可能性のある多くの操作があるために、(たとえば、多くの場合コンパイラがその操作についての情報を有しないライブラリ関数呼出しの場合)非常に非効率的な恐れがある。このような単純な技術を使用することは、多くの追加

のファイ関数を導入させる原因ともなり、SSA形式を使用するコストを劇的に増加させる恐れがある。

【0019】

本発明は、メモリ同期操作を可能な限り強化することにより、普通のプログラム構造に対して過度のオーバーヘッドなしに、非局所的記憶場所にSSA形式の使用を試みる。

【0020】

【課題を解決するための手段】

本発明において、処理手順は次のように修正される。[SSAFORM] [図12]

SSA形式内のポインタ変数を表す方法。[452]

ソース・プログラムで使用されるような単純な変数に加えて[451]、ポインタ・デレファレンスから生ずる記憶場所の参照あるいは定義も、ここで'コンプレックス変数'と呼ばれる'変数'として処理される[452]。コンプレックス変数は、ポインタ変数及びポインタからのオフセットから成る。コンプレックス変数の例は、[810]及び[820]で使用されるような、Cソース式（レフトバリュー）の' \* p 'である。

【0021】

コンプレックス変数が表す記憶場所に、コンプレックス変数[452]を同期させる適切なコピー操作を追加する方法。[図1]

これらの'コンプレックス変数'[452]は、SSA変換の間に非SSA変数として処理される[図1]（コンプレックス変数内のすべての変数レファレンスは、コンプレックス変数[452]を含む命令[440]の中のレファレンスとして処理される）。

【0022】

未知の副作用を有するすべての命令[440]を有するSSA変換されたコンプレックス変数[452]の必要なすべての同期の処理をするために、新しい段階[120]が、段階(a)[110]と段階(b)[130]の間に、SSA変換処理[図1]の中に挿入される。

## 【 0 0 2 3 】

( a' ) [ 1 2 1 ]、命令のいくつかのドミネータにより定められる' アクティブな' コンプレックス変数 [ 4 5 2 ] に未知の副作用を有する恐れのあるすべての命令 [ 4 4 0 ] に対して、変数のリスト及び可能性のある副作用 ( may\_read、may\_write ) を追加する。

## 【 0 0 2 4 】

[ 1 2 2、1 2 3 ] 次に、影響を受けた変数 [ 4 5 0 ] の S S A 変換されたバージョンが、このような副作用に対して正確に同期したことを確認するために、ライトバック [ 5 2 1 ] ( S S A 変数をその実際の位置に書き戻す)、及びリードバック ( 変数の実際の位置から新しい S S A 変数を定める) と呼ばれる特別なコピー操作を挿入する。この段階は、コンプレックス変数の同期位置からコンプレックス変数 [ 4 5 2 ] をコピー・バックすることが、その変数の新しい S S A バージョンを定める可能性がある場合には、新しいファイ関数をさらに挿入することもある。

## 【 0 0 2 5 】

ライトバック [ 5 2 1 ] 及びリードバック [ 5 2 2 ] を追加する例については、[ 図 4 ] を参照されたい。

## 【 0 0 2 6 】

## 【 発明の実施の形態 】

本発明は、コンピュータ・プログラム言語用のコンパイラへの追加であり、その基本的な制御流れを [ 図 2 ] に示す。

## 【 0 0 2 7 】

ソース・プログラム [ 3 0 1 ] は構文解析系 [ 3 1 0 ] により内部表現に変換され、最適化が使用可能であれば、内部表現は最適化プログラム [ 3 2 0 ] により最適化される。最後に、内部形式はバックエンド [ 3 3 0 ] により最終的なオブジェクト・コード [ 3 0 2 ] に変換される。S S A 形式を使用するコンパイラにおいて、最適化プログラムは少なくとも 3 つの段階を通常有する。すなわち、' S S A 前の' 内部表現から S S A 形式を使用する内部表現へのプログラムの変換 [ 図 1 ]、S S A 形式でのプログラムの最適化 [ 3 2 2 ] 及び、S S A 形式か

ら S S A 形式を使用しない内部表現へのプログラムの変換 [ 3 2 3 ] である。通常 S S A 形式は、追加的な操作の存在と表現上のある程度の制約条件のみが非 S S A 内部表現とは異なる。詳細は [ S S A F O R M ] 参照。

#### 【 0 0 2 8 】

使用されるプログラムの望ましい内部表現は次の通りである [ 図 3 ] 。

#### 【 0 0 2 9 】

プログラム [ 4 1 0 ] は一組の関数である。

#### 【 0 0 3 0 】

関数 [ 4 2 0 ] は、普通のコンパイラ概念の ' 基本ブロック ' にほぼ対応する一組の ' ブロック ' [ 4 3 0 ] である。フローグラフは、頂点がブロック [ 4 3 0 ] であり、辺がブロック [ 4 3 0 ] 間の制御流れの起こり得る転送であるグラフである。単一のブロック [ 4 3 0 ] は、 ' エントリブロック ' [ 4 2 1 ] として識別され、関数が呼び出されたとき最初に実行される関数内のブロックである。

#### 【 0 0 3 1 】

ブロック [ 4 3 0 ] の中に一連の ' 命令 ' [ 4 4 0 ] があり、それぞれの命令は単純な操作を記述している。ブロック [ 4 3 0 ] の中において、制御流れはブロック内の命令の順番と同じ順序で命令 [ 4 4 0 ] の間を移動する。制御流れの条件付きの変更は、ブロックに対する後継ブロック [ 4 3 2 ] を選択するとき、どの辺をたどるべきかを選択することによってのみ起きることがある。したがって、ブロックの中の最初の命令 [ 4 4 0 ] が実行されれば、他の命令も同様にブロック [ 4 3 0 ] 内の順序と同じ順番で実行される。

#### 【 0 0 3 2 】

命令 [ 4 4 0 ] は関数呼出しであってもよい、その場合任意の副作用の恐れがあるが、制御流れは関数呼出しに続いて命令 [ 4 4 0 ] に最終的に戻らなければならない。

#### 【 0 0 3 3 】

命令 [ 4 4 0 ] は ' 変数 ' [ 4 5 0 ] を明示的に読み取りあるいは書き込むことができる。各変数は、ソース・プログラム内の局所変数あるいは広域変数のよ



うな' 単純変数' [4 5 1] (あるいはコンパイラにより作られた一時的数値変数)、あるいは他の変数を介して間接的に参照される記憶場所を表す' コンプレックス変数' [4 5 2] のいずれかである。各変数は、変数の中にどの値を蓄積することができるかを定める型を有する。

## 【0034】

コンプレックス変数 [4 5 2] は、' \* (ベース+オフセット)' の形式であり、ここでベース [4 5 3] は変数 [4 5 0] であり、オフセット [4 5 4] は一定のオフセットである。この記号式は記憶場所 (ベース+オフセット) に蓄積された値を表す。

## 【0035】

コンプレックス変数 [4 5 2] を使用したために、計算された記憶場所から値を蓄積あるいは検索するために働く命令 [4 4 0] は通常存在しない。代わりに、ソースあるいはデスティネーションまたはその両方がコンプレックス変数 [4 5 2] である単純なコピーが使用される。同様に、すべての他の命令 [4 4 0] は、コンプレックス変数を使用して、その結果とメモリからのオペランドを蓄積あるいは検索することができる。

## 【0036】

プログラム最適化を支援するために、各関数はSSA形式に変換される。SSA形式は「従来の技術」の欄で説明されており、本発明のために修正された場合については、「課題を解決するための手段」の欄で説明されている。この変換はSSA変換と呼ばれ、3つの段階 (a)、(a')、(b) で行われる [図1]

## 【0037】

(a) [1 1 0]、[SSA FORM] で説明したように、ファイ関数が、同じ変数の複数の定義をマージすることができる関数内の任意の位置に挿入される。ファイ関数は、挿入される点において変数の新しい定義を作る。たとえば、ファイ関数 [9 1 0] は、[9 1 1] (入力プログラムの [8 2 0])、[9 1 2] (入力プログラムの [8 3 0])、さらに [1 0 1 0] において、コンプレックス変数' \* p' に書き込まれた異なる値をマージするために挿入され、[1 0

11] (入力プログラムの[820])及び入力プログラムの[830]において定められた値をマージする。

【0038】

この段階のために、プログラム内の任意の点には原変数の実在の定義は1つのみが存在する。

【0039】

(a') I. [121]、各操作に対して、どの'アクティブな'コンプレックス変数[452]が未知の副作用を有する可能性があるかを判定し、リストは操作ノートにこの情報を添付する。これらのノートは以下'可変同期'と呼ばれる。実施例のプログラムにおいて、命令[1020]、[1021]、[1022]及び[1023]は、' \* p 'をあるいは読み取りまたは修正することがある(それらの命令について何ら情報を有しない)。

【0040】

II. [122]、同時に、すべてのコンプレックス変数[452]を、元の非SSA変数(コンプレックス変数[452]に対して記憶場所である)であるそのコンプレックス変数の'同期位置'に、書き戻す必要なライトバックコピー操作[521]を追加し、コピー操作自体のデスティネーションにマークを付ける。(これは、SSA変換の段階(b)が、新しいSSA定義としてコピーのデスティネーションを処理することを防止する)。すべてのこのような'ライトバック'[521]は、対応する変数を非活動状態にし、したがって、変数が再び定義されない限り、さらにライトバック[521]されることを防止する。

【0041】

III. [123]、(同期位置に書き戻された後に)無効にされたコンプレックス変数[452]の新しいSSA定義を与えるために、必要なリードバックを追加する。

【0042】

これはデータフロー問題を解くことにより本質的に行われる。データフロー問題においては、値は'アクティブ・リードバック'である。

【0043】

アクティブ・リードバックは、上述の段階 I で決定したように、コンプレックス変数 [4 5 2] を修正することもある操作によって、あるいは、制御流れのマージ・ポイントにおける同一変数 [4 5 0] の複数のアクティブ・リードバックのマージングによって定められる。この例において、すべての関数呼出しは ' \* p ' を修正することもあり、したがって [1 0 2 0]、[1 0 2 1]、[1 0 2 2] 及び [1 0 2 4] においてリードバックによって表されなければならない。

## 【 0 0 4 4 】

アクティブ・リードバックは、アクティブ・リードバックを有するコンプレックス変数の値を使用する操作によって、あるいはその変数の他のすべてのリードバックがアクティブではない制御流れのマージ・ポイントに到達することによって参照される（そのような拡張定義は、ファイ関数を使用してコンプレックス変数の他の任意の値と次にマージされなければならないから）。

## 【 0 0 4 5 】

参照されたリードバックのみが、実際にインスタンス生成されなければならない。実施例のプログラムにおいて、唯一のインスタンス生成されたリードバックは [1 0 3 0] にある。インスタンシエーションを生ずる参照は、ソース・プログラム中の [8 4 0] における変数 ' x ' への ' \* p ' の割当てである。SSA 変換されたプログラムにおいて、この割当ては [1 0 3 0] におけるリードバックと [1 0 3 1] におけるファイ関数の間で分割される。

## 【 0 0 4 6 】

対応するコンプレックス変数 [4 5 2] の定義によって、あるいは変数の新しいリードバックによって強制終了される。この例において、次の関数呼出しが [1 0 2 2] において同一変数の新しいリードバックを定義するから、[1 0 2 1] において定義されたリードバックは強制終了される。

## 【 0 0 4 7 】

制御流れのマージ・ポイントにおいて、同一変数 [4 5 0] の他のアクティブ・リードバックとマージされ、同一変数の新しいアクティブ・リードバックを生ずる。この例において、' マージ・リードバック ' は [1 0 3 0] において定義され、[1 0 2 2] 及び [1 0 2 3] における ' \* p ' のリードバックをマージ

する。

【 0 0 4 8 】

リードバック定義の固定点に到達した後、リードバック変数 [ 4 5 0 ] の同期位置から新しい S S A 変数へ値をコピーするために、参照されたものはそれらが定義された場所で適切なコピー操作を挿入することによりインスタンス生成される。必要ならば、この新しい定義点を反映するために新しいファイ関数が挿入されてもよい。上述のように、この例において、これは [ 1 0 3 0 ] においてのみ起きる。

【 0 0 4 9 】

段階 ( a' . I ) [ 1 2 1 ] 及び ( a' . I I ) [ 1 2 2 ] は次のように行われる。

【 0 0 5 0 】

関数のエントリ・ブロック [ 4 3 0 ] の手順 ' add\_syncs\_and\_write\_backs' [ 図 5 ] を呼び出し、リストを空にするために、ACTIVE\_VARIABLES 及び ALL\_ACTIVE\_VARIABLES パラメータを初期化する。

【 0 0 5 1 】

引数 BLOCK, ACTIVE\_VARIABLES 及び ALL\_ACTIVE\_VARIABLES を有する手順 ' add\_syncs\_and\_write\_backs' は、次のように定義される [ 図 5 ] 。

【 0 0 5 2 】

[ 6 1 0 ] ブロックのすべての命令 [ 4 4 0 ] を実行する。

【 0 0 5 3 】

[ 6 2 0 ] ALL\_ACTIVE\_VARIABLES 内の各変数を実行する。

【 0 0 5 4 】

[ 6 2 1 ] 命令が変数の読み取りあるいは書き込みであれば、次に [ 6 2 2 ] ' 可変同期' を加え、予想される参照あるいは修正を命令に記述する。

【 0 0 5 5 】

[ 6 2 5 ] 命令が変数の読み取りあるいは書き込みであり、さらに ACTIVE\_VARIABLES 内にあれば、変数をその同期位置に書き戻すために、次に [ 6 2 6 ] 命令の直前に ' ライトバック' コピー操作を加え、 [ 6 2 7 ] ACTIVE\_VARIABLES

から変数を除去する。SSA変換のこの段階において、(SSA変数ではなく) 原変数のみが存在するから、このライトバック・コピー操作は変数から変数自身へのコピー ('VARIABLE := VARIABLE')により表され、デスティネーションはSSA変換されるべきではないことを示すために特別なフラグが設定される。

【0056】

[630] 命令に定義された各変数を実行する。

【0057】

ACTIVE\_VARIABLES 及び ALL\_ACTIVE\_VARIABLES に変数を加える (これらの変数の修正はこの関数に対して局所的である)。

【0058】

[650] DOMブロックにより直接支配される各ブロック [430] を実行する。

【0059】

[651] 支配されているDOMブロック上の add\_syncs\_and\_write\_backs を再帰的に使用し、ACTIVE\_VARIABLES 及び ALL\_ACTIVE\_VARIABLES の局所的な値はそれぞれ名前を付けられたパラメタとして渡される。

【0060】

段階 (a' . III) は次のように行われる [図6]。

【0061】

[701] マッピング BLOCK\_BEGIN\_READ\_BACKS 及び BLOCK\_END\_READ\_BACKS を空であるように初期化する。これらのマッピングは、フローグラフ内の各ブロックをリードバックのセットに関連づける。

【0062】

[702] 待ち行列 PENDING\_BLOCKS を関数のエントリ・ブロックに初期化する。

【0063】

[710] PENDING\_BLOCKS が空ではない時に、[711] は第一のブロック [430] をそれから除去し、そのブロック上の関数 'propagate\_block\_read\_backs' [800] を呼び出す。

## 【 0 0 6 4 】

〔 7 2 0 〕 ‘ 使用中 ’ とマークされたすべてのブロック〔 4 3 0 〕内の各リードバック R B に対して、また〔 7 2 1 〕そのソース（それがマージするリードバック）がすべて ‘ 使用中 ’ とマークされた ‘ マージ・リードバック ’ ではないすべてのブロック〔 4 3 0 〕内の各リードバック R B に対して、次のようにそのリードバックをインスタンス生成する。

## 【 0 0 6 5 】

〔 7 3 0 〕 R B が ‘ マージ・リードバック ’ であれば、リードバックの点は〔 7 4 1 〕マージが起きるブロック〔 4 3 0 〕の初めであり、他の場合には〔 7 4 2 〕リードバックを創出した命令〔 4 4 0 〕の直後である。

## 【 0 0 6 6 】

〔 7 3 1 〕 R B の変数をその同期位置から S S A 変数までコピーするリードバックの点でコピー操作を加える（上述のように、ライトバック・コピー操作の追加に対して、S S A 変数はこの段階では実際には導入されていないから、このコピー操作はその変数からそれ自身へ単純にコピーするが、S S A 変換をしないようにコピーのソースにフラグをマークする）。

## 【 0 0 6 7 】

〔 7 3 2 〕 必要であれば、新たに定義された S S A 変数を変数の他の定義とマージするために、ファイ関数を導入する。

## 【 0 0 6 8 】

パラメータブロックを有する関数 ‘ propagate\_block\_read\_backs ’ は、次のように定義される [ 図 7 ] 。

## 【 0 0 6 9 】

〔 8 0 1 〕 BLOCK\_BEGIN\_READ\_BACKS 及び、BLOCK\_END\_READ\_BACKS 内のブロックを検索し、対応するリードバック・セットに局所変数 OLD\_BEGIN\_READ\_BACKS 及び OLD\_END\_READ\_BACKS をそれぞれ割り当てる。いずれの場合にもブロックに対する項目がなければ、ブロックに適切な空の項目を追加する。

## 【 0 0 7 0 】

〔 8 1 0 〕 フローグラフ内のブロックの各先行ブロック〔 4 3 1 〕に対する終

わりのリードバックセットの共通部分を計算し、その結果である NEW\_BEGIN\_READ\_BACKS を呼び出す。共通部分は次のように計算される。

【 0 0 7 1 】

他の先行ブロックの 1 つの中に同一変数のリードバックが存在しないすべての先行リードバックは、結果から廃棄され、さらに' 参照済み' とマークされる。

【 0 0 7 2 】

所定の変数に対するリードバックが、すべての先行ブロック [ 4 3 1 ] 内の同じリードバックであれば、そのリードバックは結果に追加される。

【 0 0 7 3 】

所定の変数が少なくとも 2 つの先行ブロック [ 4 3 1 ] 内の異なるリードバックにより表されれば、すべての対応する先行リードバックを参照する' マージ・リードバック' が創出され、このマージ・リードバックが結果に追加される。

【 0 0 7 4 】

[ 8 2 0 ] NEW\_BEGIN\_READ\_BACKS が OLD\_BEGIN\_READ\_BACKS と異なっていれば、あるいはこのブロックが処理されるのが初めてであれば、

[ 8 2 1 ] NEW\_BEGIN\_READ\_BACKS を BLOCK\_BEGIN\_READ\_BACKS 内のブロックの項目として追加し、 OLD\_BEGIN\_READ\_BACKS を置換する。

【 0 0 7 5 】

[ 8 2 2 ] NEW\_BEGIN\_READ\_BACKS から NEW\_END\_READ\_BACKS を初期化する。

【 0 0 7 6 】

[ 8 3 0 ] ブロック内の各操作命令に対して、実行する。

【 0 0 7 7 】

[ 8 4 0 ] 命令内の各変数レファレンス VREF に対して、実行する。

【 0 0 7 8 】

[ 8 4 5 ] VREF が NEW\_END\_READ\_BACKS 内に項目 R B を有するなら、 [ 8 4 6 ] R B を使用中とマークし、 [ 8 4 7 ] それを NEW\_END\_READ\_BACKS から除去する。

【 0 0 7 9 】

[ 8 5 0 ] 命令内の各変数定義 VDEF に対して、実行する。

【 0 0 8 0 】

〔 8 5 5 〕 VDEF が NEW\_END\_READ\_BACKS 内に項目 R B を有するなら、〔 8 5 6 〕 R B を NEW\_END\_READ\_BACKS から除去する。

【 0 0 8 1 】

〔 8 6 0 〕 可変の変数があるいは書き込み済みであると指摘する命令内の各可変同期に対して、実行する。

【 0 0 8 2 】

〔 8 6 5 〕 NEW\_END\_READ\_BACKS への変数に対して新しいリードバック・エントリーを追加し、変数のすべての既存のリードバックを置換する。

【 0 0 8 3 】

〔 8 7 0 〕 NEW\_END\_READ\_BACKS が、OLD\_END\_READ\_BACKS と異なれば、

〔 8 7 1 〕 NEW\_END\_READ\_BACKS を BLOCK\_END\_READ\_BACKS 内のブロックの項目として追加し、 OLD\_END\_READ\_BACKS を置換する。

【 0 0 8 4 】

〔 8 8 0 〕 各ブロックの後継〔 4 3 2 〕を PENDING\_BLOCKS に追加する。

【 0 0 8 5 】

(b) 〔 1 3 0 〕 [SSA FORM] で前述のように、すべての非 SSA 変数定義がユニークな SSA 変数の定義により置換され、すべての非 SSA 変数レファレンスが適切な SSA 変数へのレファレンスにより置換される。

【 0 0 8 6 】

この規則の例外は、段階 (a') で挿入されたコピー命令〔 4 4 0 〕内で特別な'同期'位置としてマークされたコンプレックス変数〔 4 5 2 〕である。それらはそのままにしておかれ、元のコンプレックス変数〔 4 5 2 〕が参照される。

【 0 0 8 7 】

本発明を使用したものと、本発明を使用しないものの、SSA形式に変換されるプログラムの例を図 8 から図 1 1 に示す。

【 0 0 8 8 】

【発明の効果】

本発明は、別名が付けられている可能性のある状態で、非局所的メモリ位置に



対して、S S A形式の効率的な使用を可能にする同期操作を追加する。

【図面の簡単な説明】

【図 1】

本発明に使用される S S A変換処理の一般的な形式を示す。

【図 2】

コンパイラの総合的な制御流れを示す。

【図 3】

本発明の説明に使用された基本的なデータ構造を示す。

【図 4】

変数リードバック及びライトバックの配置を示す。

【図 5】

修正された S S A変換処理の段階 (a' . I) 及び (a' . I I) に対する手順の制御流れを示し、命令に変数の同期情報を追加し、変数ライトバックを関数 ' add\_syncs\_and\_write\_backs' に追加する。

【図 6】

修正された S A変換処理の段階 (a' . I I I) に対する手順の制御流れを示し、関数に変数リードバックを追加する。

【図 7】

修正された S S A変換処理 ' add\_merged\_read\_backs' の段階 (a' . I I I) により使用されたサブルーチンに対する制御流れを示す。

【図 8】

ソース・プログラムの例を示す。

【図 9】

リードバックの単純な具体化例を有する S S A変換されたプログラムを示す。

【図 1 0】

本発明において説明されたリードバックの具体化例を有する S S A変換されたプログラムを示す。

【図 1 1】

レジスタ割り当てされ S S A変換されていないプログラムを示す。

【図 1 2】

従来の S S A 変換処理の一般的な形式を示す。

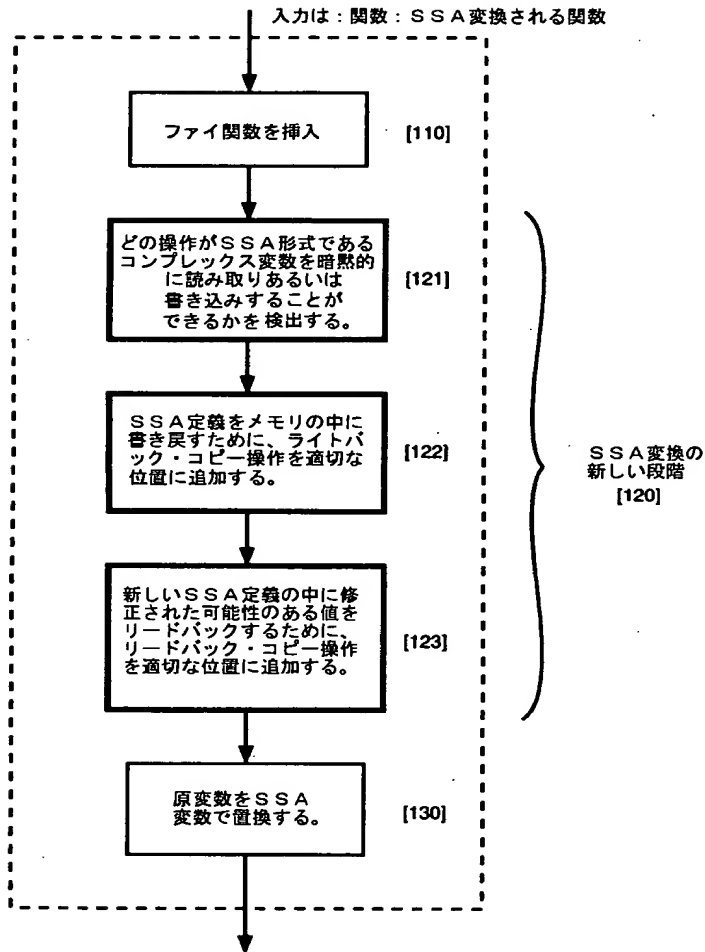
【符号の説明】

3 0 1	入力ソースファイル
3 0 2	出力ファイル
3 2 0	最適化プログラム
4 1 0	プログラム
4 2 0	関数
4 2 1	エントリブロック
4 3 0	ブロック
4 4 0	命令
4 5 0	変数

【書類名】 図面

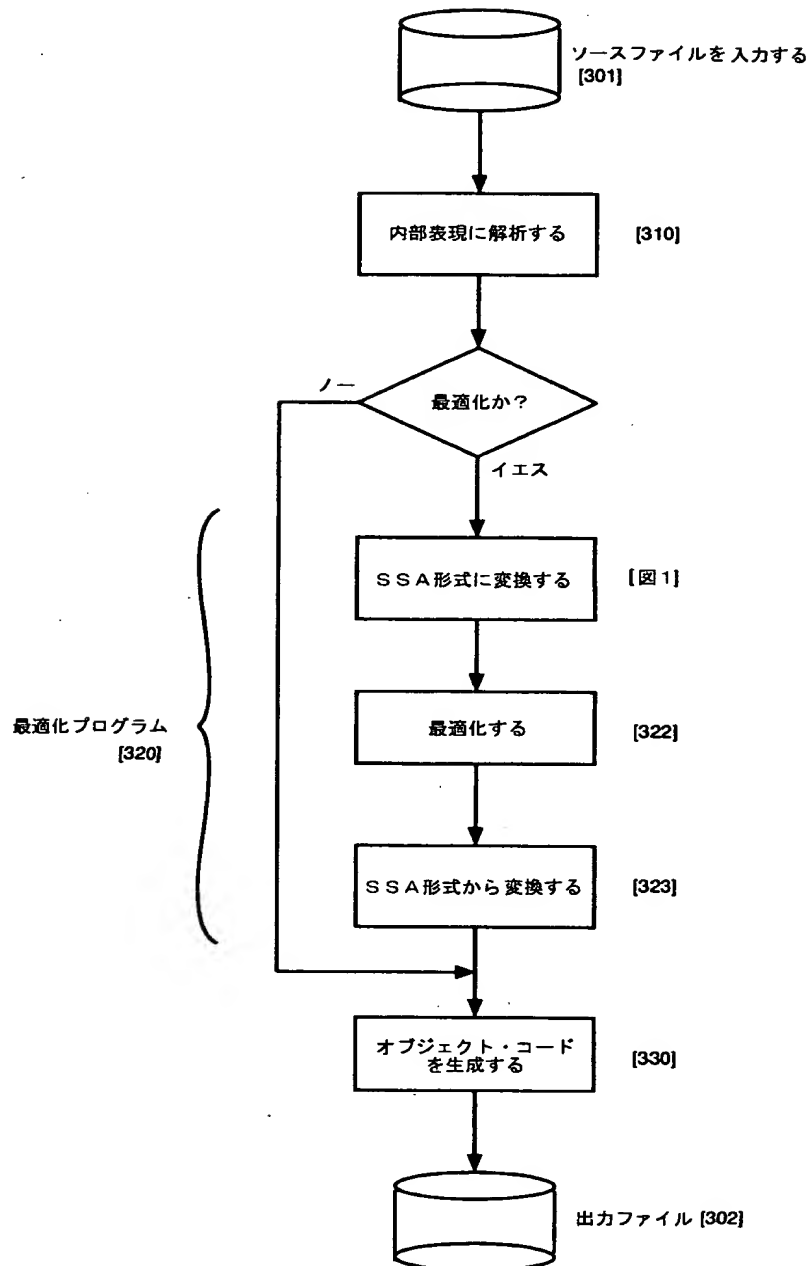
【図 1】

## 修正された SSA 変換処理



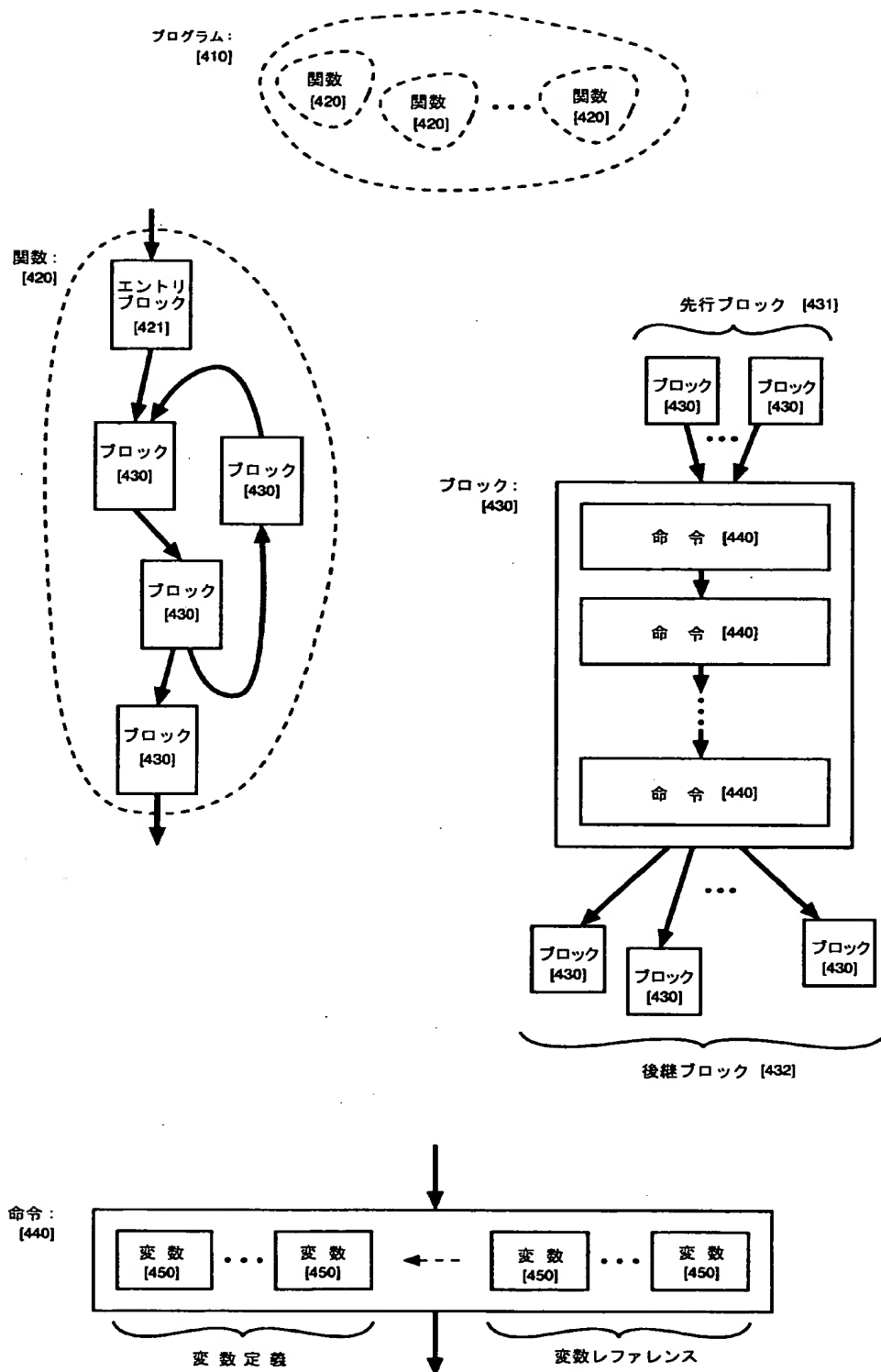
【図 2】

総合的なコンパイラ 制御流れ



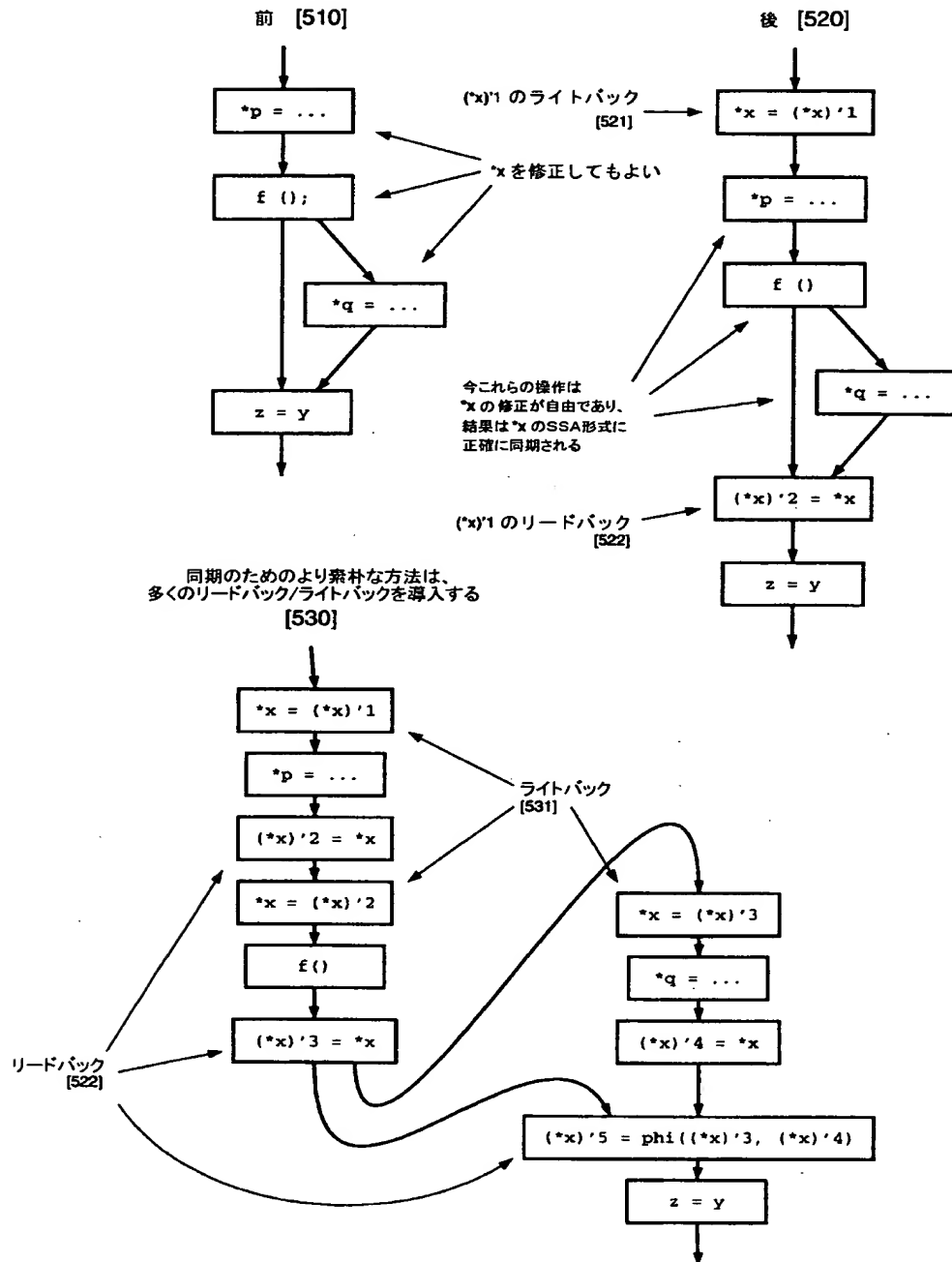
【図 3】

# プログラム表現



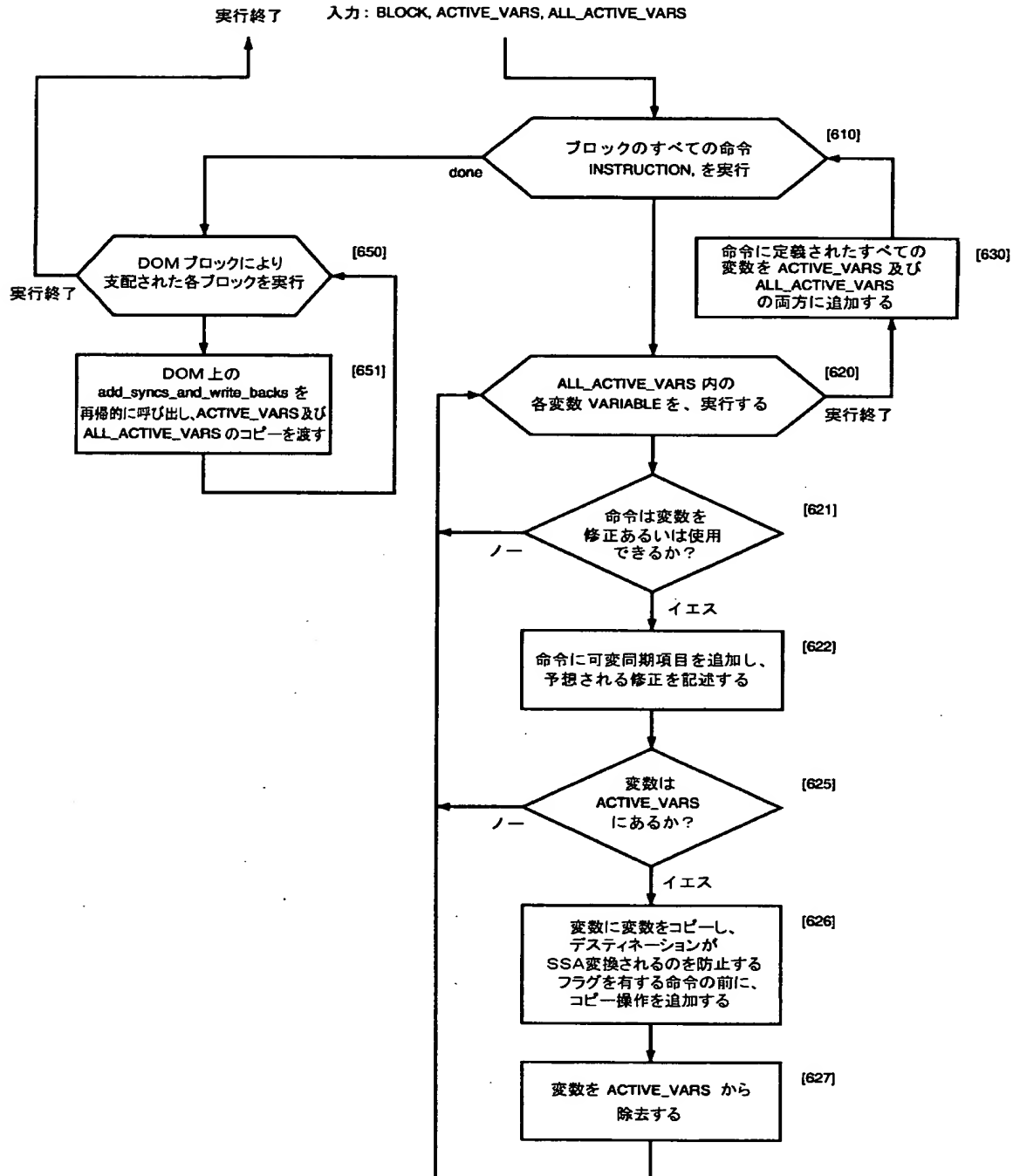
【図 4】

$*x, (*x)'1$  の SSA 形式に対するリードバック/ライトバックの配置



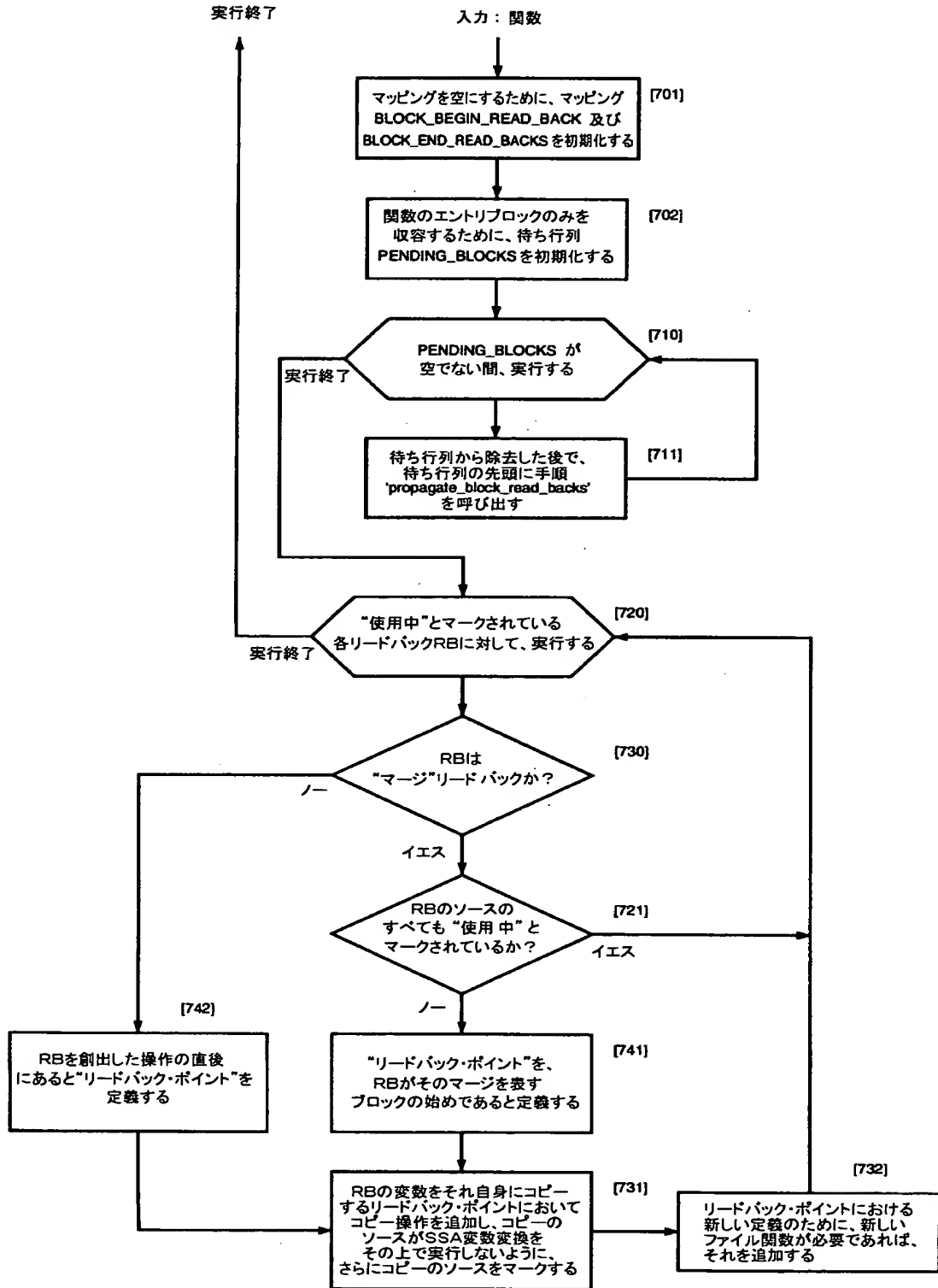
【図 5】

手順 'add\_syncs\_and\_write\_backs'



【図 6】

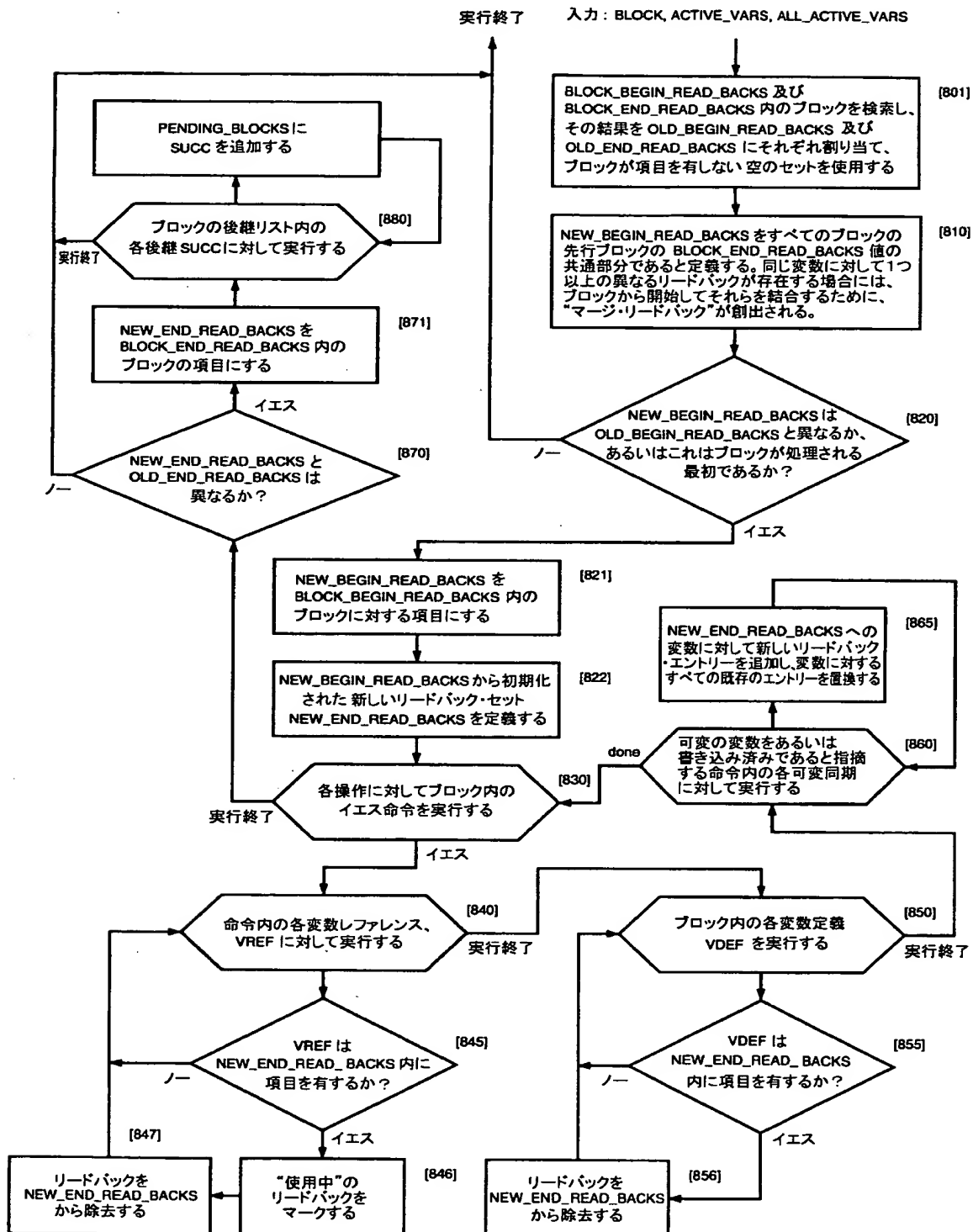
変換段階 (a'.III), リードバック の挿入





【図 7】

## 手順 'propagate\_read\_backs'



【図 8】

## ソース・プログラムの例

この短いCプログラムは本発明を説明するために 使用される :

```
extern int g (), h (), i (), x;
int foo (int *p)
{
    (*p)++;
    if (*p > 10)
    {
        g ();
        h ();
        if (x > 5)
            g ();
        if (x > 3)
            i ();
        else
            x = *p;
        *p = 5;
    }
    return *p;
}
```

[810]

やや原始的な形式に変換された同じプログラムを 次 に 示 す :

```
int foo (int *p)
{
    block1:
        *p := *p + 1;
        if (*p <= 10)
            goto block8;

    block2:
        g ();
        h ();
        if (x <= 5)
            goto block4;

    block3:
        g ();

    block4:
        if (x > 3)
            goto block6;

    block5:
        x := *p;
        goto block7;

    block6:
        i ();

    block7:
        *p := 5;

    block8:
        return *p;
}
```

[820]

[840]

[830]

## 【図 9】

## リードバックの単純な例を有する SSA 変換されたプログラム

以下は、「ファイ」操作で強化した擬似 C であり、ここで、

```
RESULT = phi (block1: VAL1, ..., blockN: VALN)
```

は、「制御流れがブロック1から来るならば、VAL1 を RESULT に割り当てる」ことを意味し、Nの各値に対して以下同様である

Nが整数である追加の変数「pvN」は、\*p の SSA バージョンであり、実際には局所変数であって、p のデレファレンスではない

```
int foo (int *p)
{
    int pv1, pv2, pv3, pv4, pv5, pv6;

    block1:
        pv1 = *p + 1;
        if (pv1 <= 10)
            goto block8;

    block2:
        *p = pv1;          /* これは PV1 を *P ヘライトバックする */
        g ();
        pv2 = *p;          /* これは *P を PV2 ヘリードバックする */
        *p = pv2;          /* これは PV2 を *P ヘライトバックする */
        h ();
        pv3 = *p;          /* これは *P を PV3 ヘリードバックする */          [912]
        if (x <= 5)
            goto block4;

    block3:
        *p = pv3;          /* これは PV4 を *P ヘライトバックする */
        g ();
        pv4 = *p;          /* これは *P を PV4 ヘリードバックする */          [911]

    block4:
        pv5 = phi (block3: pv4, block2: pv3)          [910]
        if (x > 3)
            goto block6;

    block5:
        goto block7;

    block6:
        i ();

    block7:
        x = phi (block6: x, block5: pv5);

    block8:
        pv6 = phi (block1: pv1, block7: 5);
        *p = pv6;          /* これは PV6 を *P ヘライトバックする */

    return pv6;
}
```

## 【図 1 0】

本発明で説明されたリードバックの例を有するSSA変換されたプログラム

```

int foo (int *p)
{
    int pv1, pv2, pv3;

    block1:
        pv1 = *p + 1;
        if (pv1 <= 10)
            goto block8;
                                                    [1011]

    block2:
        *p = pv1;
        g ();
        h ();
        if (x <= 5)
            goto block4;
                                                    [1021]
                                                    [1022]

    block3:
        g ();
                                                    [1023]

    block4:
        pv2 = *p;
        if (x > 3)
            goto block6;
                                                    [1030]
                                                    /* これは*PをPV2へリードバックする */

    block5:
        goto block7;

    block6:
        i ();
                                                    [1024]

    block7:
        x = phi (block6: x, block5: pv2);
                                                    [1031]

    block8:
        pv3 = phi (block1: pv1, block7: 5);
        *p = pv3;
        return pv3;
                                                    [1010]
                                                    /* これはPV3を*Pへライトバックする */
}

```

## 【図 11】

## レジスタ割付けされ、SSA 変換されていないプログラム

良いレジスタ・アロケータは短い生存期間を有する多くの変数を生成する傾向があるから、SSA形式を使用するためには、可能な限り変数をマージする  
 良いレジスタ・アロケータを有する必要がある

```
int foo (int *p)
{
    int pv;

block1:
    pv = *p + 1;
    if (pv <= 10)
        goto block8;

block2:
    *p = pv;          /* これは PVを *Pへライトバックする */
    g ();
    h ();
    if (x <= 5)
        goto block4;

block3:
    g ();

block4:
    if (x > 3)
        goto block6;

block5:
    x = *p;
    goto block7;

block6:
    i ();

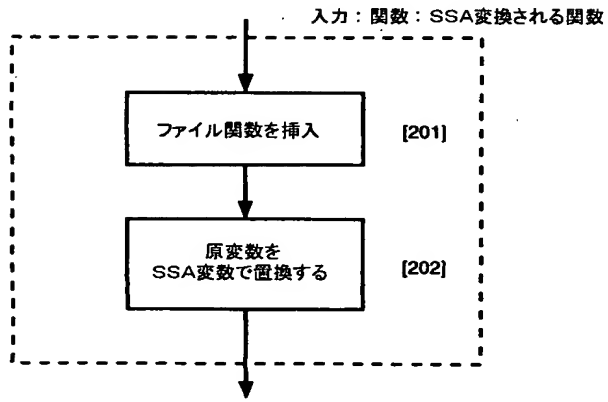
block7:
    pv = 5;

block8:
    *p = pv;          /* これは PVを *Pへライトバックする */

    return pv;
}
```

【図 1 2】

### 従来のSSA変換処理



【書類名】 要約書

【要約】

【課題】 ' S S A 形式' として知られるコンパイラ表現の通常の公式化は単に局所変数のみを処理できる。他の位置の表現を可能とするために、これを拡張することが望ましい。

【解決手段】 本発明は、別名が付けられている可能性のある状態で、非局所的メモリ位置に対して、S S A 形式の効率的な使用を可能にする同期操作を追加する。

【選択図】 図 1

出 願 人 履 歴 情 報

識別番号 [000004237]

1. 変更年月日 1990年 8月29日

[変更理由] 新規登録

住 所 東京都港区芝五丁目7番1号

氏 名 日本電気株式会社